6252

DEPARTMENT OF THE AIR FORCE
HEADQUARTERS ELECTRONIC SYSTEMS CENTER (AFMC)
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731-5000

JAN 22 1996

MEMORANDUM FOR LORAL FEDERAL SYSTEMS
ATTN: RICHARD HUGHES

FROM: ESC/ENS
5 Eglin Street
Bldg. 1704, Rm. 206
Hanscom AFB, MA 01731-2116

SUBJECT: Upgrade to Distribution Statement A

1. The STARS product, CDRL K002, "Guide to Integration of Object Oriented Methods and Cleanroom Software Engineering", is upgraded to Distribution Statement A effective 22 Jan 96.

2. Please direct any questions you may have to the Jim Henslee at (617) 377-8563.

James A. Henslee
JAMES A. HENSLEE
ESC STARS Program Manager
Software Design Center

DISTRIBUTION STATEMENT UPGRADE

CDRL Number and Task Number: CDRL I003, Task IA07

Product Title and Brief Description (what it is and what it does): "A Guide to the Integration of Object-Oriented Methods and Cleanroom Software Engineering"

The purpose of this guide is to provide the software engineering community with an Object-Oriented Cleanroom Software Process Model that can be used to support the definition or tailoring of a software development process.

Date Delivered to the Program Office: 28 Dec 95

Reviewer's Name, Extension Number and Date of Review: Marcelle Nachef, (617) 377-4918, 22 Jan 96

Intended Audience: Public conferences, trade shows and workshops

Comments:

The STARS product, A Guide to the Integration of Object-Oriented Methods and Cleanroom Software Engineering, previously under Distribution Statement C, is upgraded to Distribution Statement A effective 22 Jan 96. This product is generic and does not apply to specific defense articles and defense services. In accordance with Memorandum of Agreement between ESC/PA and ESC/ENS concerning upgrades of STARS products to Distribution A, the STARS program office at ESC/ENS has reviewed this product and has determined that the information is unclassified, technically accurate, and suitable for public release.

James A. Henslee
Approving Authority

1/22/96
Date

Task/Subtask IA09
CDRL Sequence K002-001
29 February 1996

Software Technology for Adaptable, Reliable Systems (STARS) Program

A GUIDE TO INTEGRATION OF OBJECT-ORIENTED METHODS AND CLEANROOM SOFTWARE ENGINEERING

Contract No. F19628-93-C-1029

Prepared for:

Electronic Systems Center
Air Force Material Command, USAF
Hanscom AFB, MA 10731-2116

Prepared by:

Loral Federal Systems
700 North Frederick Avenue
Gaithersburg, MD 20879

Preface

This document was developed by Loral Federal Systems-Gaithersburg (LFS-G), located at 700 North Frederick Avenue, Gaithersburg, MD 20879, Software Engineering Technology, Inc., Mills Building, Suite 304, 2200 Sutherland Avenue, Knoxville, TN 37919, and the University of Tennessee, Computer Science Department, Software Quality Research Laboratory, Ayres Hall, Knoxville, Tennessee. The authors of this document are William H. Ett of Loral Federal Systems (ettb@lfs.loral.com) or Dr. Carmen Trammell of the University of Tennessee (trammell@cs.utk.edu). Questions or comments regarding this document should be sent to William H. Ett of Loral Federal Systems.

This document has been reviewed by Mr. Grady Booch of the Rational Corporation, Mr. Steve Mellor and Mr. David Murley of Project Technology, Incorporated, and Ms. Susanne Dyrhage, Mr. Magnus Christerson and Johan Aronsson of Rational AB (formerly Objectory AB). Comments received from the final document review cycle were incorporated in this final document deliverable.

A Guide to Integration of Object-Oriented Methods and Cleanroom Software Engineering

EXECUTIVE SUMMARY

Task

The results of object-oriented software development projects in the DoD have been disappointing relative to the promise that OO technology seems to hold. STARS Task IA09 was commissioned to explore the feasibility of combining object-oriented methods (well known for their focus on reusability) with Cleanroom software engineering (well known for its emphasis on reliability) to define a software process capable of producing results that are not only reusable, but predictable and of high quality.

Approach

A generic software process was defined in terms of phases, activities, and work products. Three OO processes (Booch, Objectory, Shlaer-Mellor) and the Cleanroom process were outlined by phases, activities, and work products as well, using a documentation baseline given by the methodologists. The three OO methods and Cleanroom were compared to the generic process for thoroughness and coverage, and an integrated process was defined.

Findings

1. The common strength of object-oriented methods is their emphasis on domain analysis, i.e., identifying abstractions and relationships that are characteristic of applications in the domain. Cleanroom lacks the emphasis on strategic reuse in object-oriented methods.
2. The strength of the Cleanroom approach is its rigor in specification, design, verification, and testing in an application project. OO methods generally lack the engineering, mathematical, and statistical formalisms of the Cleanroom process.
3. OO and Cleanroom are compatible with respect to the essential definition of an object in terms of (1) external behavior and (2) internal data and access programs. Given this fundamental compatibility, the differences in the methods can be complementary.
4. Cleanroom is the process most amenable to software development under statistical process control because of its combination of an iterative process (incremental development) and statistical testing.

Recommendations

Use OO for front-end domain analysis, and Cleanroom for life cycle application engineering. Use OO for exploring the problem, and Cleanroom for developing the solution. Use OO to identify parts, and Cleanroom to develop the whole. Look for breakthrough concepts from OO, and breakthrough performance from Cleanroom.

Specifically, use OO for identifying the domains that are pertinent to a problem and characterizing the objects and object relationships in each domain. Use Cleanroom for formal specification, design, verification, usage modeling, and usage testing through incremental development under statistical process control.

I. Introduction

Description of STARS Task IA09

The purpose of the task was to provide guidance for software policy makers, managers, and practitioners about complementary use of object-oriented methods (as given by Booch, Jacobson et al., and Shlaer-Mellor) and the Cleanroom software engineering process.

The work in this task was based on the assumptions that

- object-oriented methods support domain-specific reuse,
- the Cleanroom process supports high reliability, and
- object-oriented and Cleanroom ideas are compatible and complementary.

In contrast to the many studies that have compared features and techniques of object-oriented methods (see Documentation Baseline below), the candidate methods in this task were examined from a life cycle perspective.

Purpose of this Guide

The purpose of this Guide is to provide the software engineering community with an Object-Oriented Cleanroom Software Process Model that can be used to support the definition or tailoring of a software development process. The Guide suggests ways in which OO methods might enhance the analysis phase and reuse potential in Cleanroom, and that Cleanroom might improve the way software objects are specified, verified, and certified.

Audience for the Guide

This Guide is intended for software policy makers, software managers, task leaders, practitioners, and process engineers.

How to Use the Guide

Policy Makers, Software Managers

Policy makers and managers may be most interested in the overall perspective on object-oriented methods and Cleanroom given in the Executive Summary and the Conclusions. In general, these sections recommend OO for domain analysis and Cleanroom for application engineering. Other uses of the Guide may be as follows.

- **Orientation**

The Guide may be used to obtain an overview of the methods. Each method is described as a set of phases, activities, and work products.

- **Risk Reduction**

Relative life cycle emphases are shown in the mappings of Cleanroom to each OO process and of the OO and Cleanroom processes to the Generic Process. These mappings and the associated discussion may help the policy maker or manager identify risks that must be addressed in selecting and planning project methods.

Task Leaders, Practitioners, and Process Engineers

Task leaders, practitioners, and process engineers responsible for defining software processes may be most interested in the descriptions of and mappings between individual methods. Specific uses of this information may be as follows.

- **Evaluation**

Mappings between and among the methods are given in tables. Readers can determine which aspects of the software life cycle are (and are not) covered in each method. The Guide provides a roadmap based on the software life cycle that may aid in evaluation of methods.

- **Extension**

Using each OO method as a "base process," Cleanroom extensions to the base process are given. The extensions represent strengths of the Cleanroom method that are not emphasized in the base method.

- **Integration**

A Generic Object Oriented Cleanroom process is defined using components of the individual methods. The integrated process draws on strengths of the methods studied in this task.

Conventions Used in This Guide

Process Descriptions

- The method author's own words have been used for names and definitions to the maximum degree possible.
- An explicit representation of iteration and concurrency in processes is not used. Aspects of an incremental approach to development are used in all the methods. The listing of phases and activities is this Guide is not intended to imply a sequential life cycle.
- Phases and activities in the process descriptions are given in bold, and work products are given in italics.

Mapping of Methods to Generic Process

- Strongly mapped phases and activities appear as normal bold text; strongly mapped work products are in bold italic text. See the process descriptions for definitions of process parts.
- Weakly mapped process parts appear as normal text; weak mappings include both explicit process parts and ideas "mentioned" in source documents but not explicitly specified as an activity or product.

Documentation Baseline for Booch, Objectory, and Shlaer-Mellor

Booch Method

Booch, G. Object-Oriented Analysis and Design with Applications, 2nd edition. Benjamin-Cummings, 1994.

Booch, G. Object Solutions: Managing the Object-Oriented Project. Addison Wesley, 1996.

Objectory Method

Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992. Also, revised 4th printing, 1993.

Objectory AB, Objectory: Project Management, Version 3.6, 1995.

Objectory AB, Objectory: Configuring your Process, Version 3.6, 1995.

Objectory AB, Objectory: Requirements Analysis, Version 3.6, 1995.

Objectory AB, Objectory: Robustness Analysis, Version 3.6, 1995.

Objectory AB, Objectory: Design, Implementation and Testing, Version 3.6, 1995.

Shlaer-Mellor Method

Lang, N. "Shlaer-Mellor Object-Oriented Analysis Rules." Software Engineering Notes, Project Technology Technical Report, January 1993.

Montrose, R. "Object-Oriented Development Using the Shlaer-Mellor Method." Project Technology Technical Report, 1994.

Shlaer, S. and N. Lang. "Shlaer-Mellor Method: The OOA96 Report." Project Technology Technical Report, 1995.

Shlaer, S. and S.J. Mellor. Object-Oriented Systems Analysis: Modeling the World in Data. Prentice-Hall, 1988.

Shlaer, S. and S.J. Mellor. Object-Oriented Lifecycles: Modeling the World in States. Prentice-Hall, 1992.

Shlaer, S. and S.J. Mellor. "The Shlaer-Mellor Method." Project Technology Technical Report, 1993.

Shlaer, S. and S. Mellor. "Real-Time Recursive Design." Project Technology Technical Report, 1992.

Shlaer, S., S. Mellor, and D. Grand. "The Project Matrix: A Model for Software Engineering Project Management." Proceedings of the Third Software Engineering and Standards Application Workshop, October 1984.

Shlaer, S., S. Mellor, and W. Hywari. "OODLE: A Language-Independent Notation for Object-Oriented Design." Project Technology Internal Report, 1994.

Cleanroom Documentation

Linger, R. C., H. D. Mills and B. I. Witt. Structured Programming: Theory and Practice. Reading, MA: Addison-Wesley, 1979.

Mills, H. D. "Stepwise Refinement and Verification in Box-Structured Systems." *IEEE Computer*, June 1988, pp. 23-36.

Mill, H.D., R.C. Linger, and A.R. Hevner. Principles of Information Systems Analysis and Design. New York: Academic Press, 1986.

Whittaker, J. A. and J.H. Poore. "Markov Analysis of Software Specifications." *ACM Transactions on Software Engineering and Methodology*, January 1993.

Whittaker, J. A. and M. G. Thomason. "A Markov Chain Model for Statistical Software Testing." *IEEE Transactions on Software Engineering*, October 1994, pp. 812-824.

Software Quality Research Laboratory. "Cleanroom Process Description." Department of Computer Science, University of Tennessee, June 1995.

Comparisons of Object-Oriented Methods - A Short Bibliography

de Champeaux, D. and P. Faure. "A Comparative Study of Object-Oriented Analysis Methods." *Journal of Object-Oriented Programming*, March/April 1992.

Fichman, R. and C. Kemerer. "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique." *IEEE Computer*, October 1992, pp. 22-39.

Hong, S. and B. Koelzer. "A Comparison of Software Reuse Support in Object-Oriented Methodologies." IRMA '95.

Hong, S., G. van den Goor, and S. Brinkkemper. "A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies." Proceedings of the 26th Hawaii International Conference on System Sciences, January 1993.

Martin, J. and J. Odell. Object-Oriented Methods, A Foundation. Prentice Hall, 1995.

Mellor, S. "A Comparison of the Booch Method and Shlaer-Mellor OOA/RD." Project Technology Internal Report, 1993.

Jacobson, I., et al. Object Oriented Software Engineering: A Use Case Driven Approach, Chapter 16 -- "Other object-oriented methods." Addison-Wesley, 1992.

Monarchi, D. and G. Puhr. "A Research Typology for Object-Oriented Analysis and Design." *Communications of the ACM*, Volume 33, Number 9, September 1992.

Shlaer, S. "A Comparison of OOA and OMT." Project Technology, Inc., Technical Report, 1992.

ARPA STARS (Software Technology for Adaptable, Reliable Systems) Program. "Cleanroom Software Engineering: A DoD STARS Tutorial by Software Engineering Technology, Inc." http://source.asset.com/stars/loral/cleanroom/tutorial/cleanroom.html

Acknowledgments

The authors thank Don MacFarlane, Stacy Prowell, and Steve Zevan of the University of Tennessee Software Quality Research Laboratory for their draft descriptions of the object-oriented methods.

The authors also thank Ivar Jacobson and Magnus Christerson of Rational AB, Grady Booch of Rational, and Steve Mellor, Rod Montrose, and Dave Murley of Project Technology, Incorporated for the materials and assistance they provided to support the task. We also thank STARS Program Manager, Linda Brown, for her sponsorship of this task.

II. Generic Process

Overview

The Generic Process is a composite of many models of the software life cycle. It was created for this Guide, and is used as a background for understanding the life cycle emphases and coverage of the specific processes examined in this task.

The Generic Process is based on the following themes:

- front-end domain understanding
- bottom-up identification of domain objects and their relationships
- characterization of expected usage
- top-down system specification and architectural design
- incremental development and testing

The Generic Process is presented as a set of life cycle phases, activities, and work products, as are the other processes in the Guide. In the composite view of OO and Cleanroom processes in the final section of this Guide, the Generic Process is used as a baseline, and all other processes are mapped to it.

Generic Software Process - Glossary

architecture - a description of the organization and interfaces of all components of a system, from high-level abstractions (subsystems) to low-level abstractions (modules).

constraint - a requirement for a system that constrains the way in which the system will be designed and implemented.

implementation-independent software design - a representation of how a system will satisfy its functional requirements without regard to the target implementation environment.

increment - a period of time in which a specified unit of work will be performed; usually, an iteration of the development process in which a specified set of system functions will be implemented.

operation - a function performed by the system at the request of a system user or a system object.

problem domain - the environment or "world" in which a system operates.

requirements - the functions that a system must perform and the constraints on how those functions must be performed.

response - an output produced by a system or system object upon receipt of a stimulus.

scenario - a description of a sequence of stimulus-response transactions between a user and a system.

specification - a relation between all external stimuli to an object and the expected external responses to those stimuli in all circumstances of use; an implementation-independent view of what a system is to do.

stimulus - an input to a system or system object.

system object - a system, subsystem, module, or any other component in a system.

GENERIC SOFTWARE PROCESS - PHASES

1. Concept Definition

The purpose of this phase is to define the system's purpose in the context of the mission it is intended to support. Describing the mission for a proposed system provides all stakeholders with a vision of who will use the system, what it will do, and how it is intended to be used. This phase may be supported by proof-of-concept prototyping.

2. System Analysis

The purpose of this phase to understand and define the requirements for a system and the domain in which it will operate. The activities typically performed in support of system analysis are:

- Describing the problem domain
- Defining or clarifying customer requirements
- Identifying and assessing risks
- Identifying reuse opportunities
- Preparing a plan for the top-level specification and design

3. System Specification

The purpose of this phase is to define the implementation-independent behavior that a software system is to exhibit. This is done by identifying the stimuli the system must accept and the responses it must produce. The activities typically performed in support of system specification are:

- Specifying the user interface
- Implementing user interface prototypes
- Identifying stimuli and responses for the system on the basis of the requirements
- Preparing scenarios of system usage as sequences of system transactions
- Preparing the system specification

4. System Design

The purpose of this phase is to define how the system will be built. The activities typically performed in support of system design are:

- Defining an implementation-independent design that satisfies the functional requirements for a system
- Designing candidate system architectures that address constraints

- Preparing prototypes or simulations to demonstrate that a system architecture can address critical transaction sequences and constraints
- Selecting a candidate architecture to support system implementation

5. System Implementation

The purpose of this phase is to build the system. The activities typically performed during this phase are:

- Preparing an incremental development plan
- Developing a software increment
- Testing a software increment
- Assessing progress and risks after each increment

6. System Acceptance

The purpose of this phase is to determine whether the system satisfies the requirements and constraints. System acceptance is usually supported by operational test and evaluation.

Not in the scope of this study.

7. System Maintenance and Evolution

The purpose of this phase is to correct or modify the function of the system over its life.

Not in the scope of this study.

Generic Software Process - Activities and Work Products

1. Concept Definition

1.1 Define the Mission

Statement of why the system is being built and what it is intended to do.

Mission Statement - A brief description of the mission a proposed software system is to satisfy, who the users of the proposed system are, and how they will use it. The mission statement states the purpose and high level requirements for the system. It provides all project members with a vision of the system they are to create and the mission it is intended to support.

2. System Analysis

2.1 Analyze Problem Domain

Characterization of the system's "world": objects, relationships, behavior, boundaries.

Domain Description - a description of the environment in which the proposed software system will operate: the objects in the environment, the characteristics of those objects, and the communication patterns among the objects.

2.2 Analyze Requirements

Development of the list of "shalls": the functional requirements and constraints for the system.

Requirements Statement - a list of the functional requirements for and constraints on a proposed software system; constraints may relate to performance, capacity, timing, quality, portability, etc.

2.3 Plan Specification and Design Activities

Risk assessment, reuse planning, and planning for the (possibly iterative) top-level system specification and design phase. Domain-specific assets and COTS products are identified for prospective reuse.

Specification and Design Plan - a definition of the activities necessary to prepare a complete specification of the external behavior of the software system and an implementation-independent design.

2.4 Review Analysis Phase Work Products

Confirmation that requirements and plans are complete, and are understood and accepted by both developer and customer. Also, examination of the System Analysis process for lessons learned about process improvement.

Analysis Review Criteria - criteria for team review of all work products.

3. System Specification

3.1 Specify User Interface

Representation of all external interfaces with human, hardware, or software users.

User Interface Specification - the specification of how an external agent will use the software, where an external agent may be a human user or an external system. The User Interface Specification represents all possible system inputs and outputs, such as screens, displays, reports, prompts, data formats, communication protocols, sensors, signals, or any other interface between the system and its users.

User Interface Prototype(s) - a prototype built to demonstrate and clarify a system's concept of use with its users.

3.2 Describe Usage Scenarios

Identification of all external stimuli and responses, and scenarios of their use.

External Stimulus and Response List - a list that identifies the stimuli and responses associated with each system requirement. The External Stimulus and Response List summarizes the stimuli and responses represented in the User Interface Specification.

Usage Scenarios - sequences of stimuli and responses that describe system usage scenarios. Usage scenarios describe the sequence of transactions from initiation to conclusion of a system function. The usage scenarios provide an informal specification of the primary uses of the system.

3.3 Specify Software System

Representation of external system stimuli, responses, and functions in a formal notation.

Software Specification - the expected external behavior of the system in terms of all possible responses to all possible sequences of stimuli. The Software Specification addresses functional requirements in the Requirements Statement, but does not address constraints.

3.4 Review Specification Phase Work Products

Confirmation that the system specification is correct, complete, and consistent, and that it is understood and accepted by all members of the project team. Also, examination of the System Specification process for lessons learned about process improvement.

Specification Review Criteria - criteria for team review of all work products, and team verification of the Software Specification against the Requirements Statement.

4. System Design

4.1 Identify Logical System Objects

Invention of objects to support transitions between system stimuli and system responses.

Logical Object Catalog - definition of primary internal logical system objects. Each object is defined in terms of its interface, data, and access functions.

4.2 Prepare Implementation-Independent Software Design

Representation of communication among the high-level system objects to accomplish functional requirements.

Implementation-Independent Software Design - an organization of all the high-level logical objects needed to implement the Software Specification. The objects are organized in terms of their relationships to each other (e.g., instantiating, containing) and their sequential and concurrent accesses of each other.

4.3 Develop Software Architecture

Analysis of trade-offs; prototyping; allocation of function, performance, and reliability requirements to components.

Candidate Software Architecture(s) - the Implementation-Independent Software Design organized as a set of physical components which address system constraints. The architecture shows the dependencies and interfaces among software components.

Software Architecture - a Candidate Software Architecture that is capable of satisfying system constraints and has been selected for the software system.

4.4 Specify System Objects

Performance of System Analysis and System Specification phases (above), as needed, for subsystems.

Software Object Specifications - the expected behavior of each subsystem in terms of all possible responses to all possible sequences of stimuli to each subsystem. Subsystem stimuli and responses may include both external events associated with system functions allocated to the system object and internal events associated with abstractions created during system design.

The design of each system object is performed as an implementation activity. The implementation of a system object may be performed as a separate development project.

4.5 Review of Design Phase Work Products

Confirmation that the Software Architecture is a correct implementation of the Software Specification. Also, examination of the System Design process for lessons learned about process improvement.

Design Review Criteria - criteria for team review of all work products, and team verification of Software Architecture against the functional requirements in the Software Specification and the constraints in the Requirements Statement.

5. System Implementation

5.1 Plan Increments

Risk assessment, reuse planning, and planning for an iterative cycle of incremental system implementation based on the top-level Software Specification and Software Architecture.

Incremental Development Plan - a plan for building and testing a software system as a set of increments. The plan defines the content of and schedule for the increments. At the completion of each increment the plan is reviewed and revised based on work-to-date and identified risks.

5.2 Develop Increment

Stepwise specification, design, and verification of a system increment.

Implemented Increment - A partially (or completely) implemented software system that has been specified, designed, coded, inspected, and is ready to be tested.

5.3 Test Increment

Testing of the system increment.

Tested Increment - A partially (or completely) implemented software system that has been tested, has met some quality criterion, and may be demonstrated to the customer.

5.4 Review Increment Work Products

Confirmation that the Implemented Increment performs correctly according to the Software Specification and Requirements Statement. Also, examination of the System Implementation process in the increment for lessons learned about process improvement.

Increment Review Criteria - criteria for team verification of the correctness of the Implemented Increment, team review of the performance of the software during testing, and team review of opportunities for process improvement.

6. System Acceptance

Not in the scope of this study.

7. System Maintenance and Evolution

Not in the scope of this study.

III. CLEANROOM PROCESS

Overview

In the Cleanroom software engineering process, engineering formalisms underlie incremental development; mathematical formalisms underlie specification, design, and correctness verification; and statistical formalisms underlie certification testing. These mature formalisms afford the predictability that is lacking in the comparatively heuristic object-oriented approaches.

Erroneously associated with functional decomposition, Cleanroom lends itself to object-orientation without difficulty. A Cleanroom component is defined in a black box view (an object's external behavior), a state box view (an object's encapsulated data), and a clear box view (services that process external requests and access encapsulated data). A Cleanroom component is an object in the most technical sense.

Cleanroom lacks the emphasis on domain analysis in object-oriented methods. The common strength of object-oriented methods is their pursuit of abstractions and relationships that are characteristic of applications in the domain. The Cleanroom process would be enhanced by an object-oriented front-end domain analysis.

For more information about Cleanroom, see "Cleanroom Software Engineering: A DoD STARS Tutorial by Software Engineering Technology, Inc." at

http://source.asset.com/stars/loral/cleanroom/tutorial/cleanroom.html

CLEANROOM - GLOSSARY

black box - A specification for the external behavior of an object. The black box accepts stimuli, produces responses, and specifies the functional relationship between stimulus histories and responses.

box structure - A data abstraction defined in three forms: a history-based specification (the black box), a state-based specification (the state box), and an implemented procedure (the clear box).

certification - The conduct of software testing as a statistical experiment and subsequent quantification of software reliability.

clear box - Full procedural design, specifying both data flow and control flow; new black boxes may be created to encapsulate lower-level functions.

common service - A reusable box structure.

increment - A complete iteration of the development and testing activities.

incremental development - A top-down approach to development in which a software system is developed and tested as a succession of cumulative subsets of system function.

referential transparency - Complete specification of the behavior of new black boxes in a clear box; enables independent development of the black boxes without need for further information.

reliability - The probability that a system will perform successfully in a randomly-selected scenario of use from the population of uses represented by the usage model.

specification - The black box functional mapping of all possible stimulus histories to their correct responses.

state - The encapsulation of stimulus history.

state box - A specification of the behavior of a data abstraction as a mathematical function where the domain is all possible (stimulus, old state) pairs and the range is all correct (response, new state) pairs.

state migration - "Information hiding;" the placement of state data at the lowest possible level in a system.

statistical quality control - Measurement of performance in each increment, and comparison of actual performance with preestablished standards to determine whether or not the development process is "in control."

test case - A complete usage scenario that is randomly generated from the usage model; a sequences of user inputs.

test script - Test cases and instructions to testers on test execution and validation.

transaction closure - Identification of the complete set of stimulus, response, and state data items needed to satisfy system requirements for all possible users and all possible uses.

usage model - A formal structure representing the operational capabilities of a system (i.e., what users can do) and expected operational use (i.e., what users are likely to do).

verification - Confirmation that a software design is a correct implementation of its specification.

Reference

"Cleanroom Process Description." Software Quality Research Laboratory, Department of Computer Science, University of Tennessee, Knoxville, Tn., June 1995.

CLEANROOM - PHASES

1. Project Management

A Cleanroom project is conducted through a process for management and technical control of incremental software development and certification. Project management activities are ongoing during a project, and are concurrent with Increment Management. Project management activities include Customer Interaction, Process Control, Schedule Development and Maintenance, Training, and Tasking.

2. Increment Management

A Cleanroom project is managed as a series of accumulating increments. A Cleanroom Increment is a complete development and certification cycle in a Cleanroom project. The scope of the work in an increment includes work done in previous increments plus the new function in the current increment.

2.1 Top-Level Specification and Design

The first increment in a Cleanroom project begins with creation and review of top-level artifacts (the Top-Level Box Structure Specification, Top-Level Box Structure Design, and Usage Specification). In subsequent increments, these artifacts are updated as needed.

2.2 Increment Planning

An Incremental Development Plan is developed for the Cleanroom project based on the project schedule, the top-level artifacts, and the capability of the team. At the end of each increment, the plan is reviewed and updated to reflect changes in schedule, specifications, design, or evaluation of team capability.

2.3 Box Structure System Decomposition

The Box Structure method of specification and design, which involves stepwise refinement and functional verification, is used for system decomposition. Individuals draft an assigned system part, and then individual work undergoes team design review and correctness verification until it is accepted by the full team. Intellectual control of the evolving system is maintained through adherence to the Box Structure method and the ongoing communication that occurs in peer review.

2.4 Statistical Test Planning

The Top-Level Box Structure Specification and the Usage Specification are used to develop a usage model and test plan.

2.5 Increment Certification

Certification of an increment begins when a complete and verified increment is delivered for testing. The increment is placed under engineering change control. The test plan is executed. Failure data and estimates of reliability are maintained throughout certification testing. Testing of an increment stops when either (1) customer requirements for reliability have been achieved, or (2) process control standards have been violated. Certification of the final increment constitutes system certification.

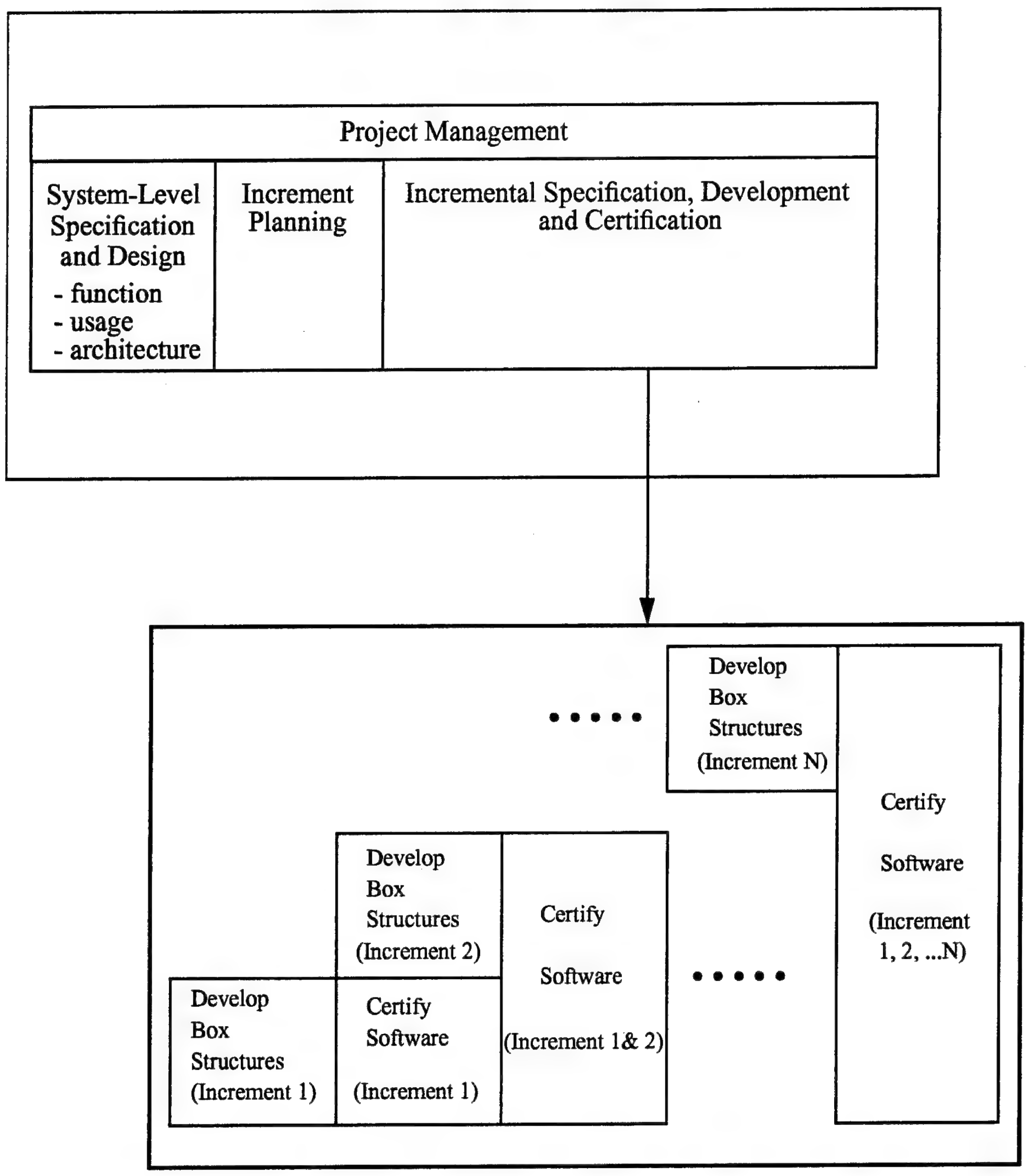
2.6 Increment Conclusion

When certification testing has been completed, project records are compiled to form the increment report. The team reviews the process as performed in the increment against the intended process to identify opportunities for process improvement.

Figure 4 illustrates the Cleanroom Software Engineering Process as described in the "Cleanroom Process Description."

Reference

"Cleanroom Process Description." Software Quality Research Laboratory, Department of Computer Science, University of Tennessee, Knoxville, Tn., June 1995.


Project Management
System-Level Specification and Design
- function
- usage
- architecture
Increment Planning
Incremental Specification, Development and Certification
Develop Box Structures (Increment N)
Certify Software (Increment 1, 2, ...N)
Develop Box Structures (Increment 2)
Certify Software (Increment 1& 2)
Develop Box Structures (Increment 1)
Certify Software (Increment 1)

Development and Certification of a Pipeline of Software Increments

Figure 4: Cleanroom Software Engineering Process

CLEANROOM - ACTIVITIES AND WORK PRODUCTS

1. Project Management

Process References - process documentation: development and testing policies, review protocols for specification, design, verification, and usage modeling; data collection forms for customer interaction, tasking, verification, compilation, and testing; Fault Classification Scheme; and Specification and Development Style Guide.

Project Records - technical references, and design and certification history for related work.

1.1 Customer Interaction

The project manager interacts with the customer on matters of the Customer Requirements, Project Schedule and Incremental Development Plan. Development team members may interact with the customer to clarify Customer Requirements related to function. Certification team members may interact with the customer to gather data concerning Customer Requirements related to expected usage. Each meeting with the customer is documented.

Customer Requirements - the functional, performance, implementation, usage, reliability, and schedule requirements for a software system to be developed under the Cleanroom process.

Customer Interaction Records - completed Customer Interaction Meeting Forms which contain all information about meetings with customers.

1.2 Process Control

During verification, compilation, and testing, team members review and compare performance with Process Control Standards. If performance meets the standards, the process continues as planned. If performance on any activity does not meet the standards, a team review is held to discuss the implications. A decision is made to either continue work and review status after a specified interval, or to roll back the process and redo some task(s).

Process Control Standards - the reference values for measures of process control.

Process Control Meeting Records - completed Process Control Meeting Forms which contain information about meetings held to determine if the process is in control.

1.3 Schedule Development and Maintenance

The Project Schedule is set and revised based on Customer Requirements and project resources. The Project Schedule contains milestones for staffing, budgeting and reporting. The schedule may be revised as a result of customer interaction and evolution of the Incremental Development Plan.

Project Schedule - major milestones related to staffing, budget, customer reports and reviews, and incremental development, and more specific timetables for development and certification.

1.4 Training

The orientation of new team members occurs through an initial overview by the project manager, and through "just-in-time" training by co-workers throughout the project using Process Documentation.

1.5 Tasking

The project manager assigns the roles of chief specification engineer, specification team members, chief designer, development team members, chief certification engineer, and certification team members. The chief designer assigns development tasks to development team members. The chief certification engineer assigns usage modeling, test planning, and certification tasks to certification team members.

Tasking Records - completed Tasking Forms which contain information about each team member and his/her role in the current project.

2. Increment Management

2.1 Top-Level Specification and Design

2.1.1 Top-Level Box Structure Specification Development

The Top-Level Box Structure Specification is developed by the full team using Customer Requirements. Clarification of requirements, if needed, is obtained through customer interaction. The specification is represented in the format given in the Specification and Development Style Guide, and involves reference to Common Specification Function Documentation. New common specification functions may be created or existing ones updated.

Top-Level Box Structure Specification - a complete representation of the external view (i.e., the user's view) of the system. It includes the top-level black box in the Box Structure usage hierarchy.

2.1.2 Top-Level Box Structure Specification Review

The Top-Level Box Structure Specification is reviewed by the full team using the Specification Review Protocol. It is reviewed with reference to Customer Requirements and other reference documents as needed. If changes in the specification are necessary or desirable, another iteration of Specification Development and Review occurs.

2.1.3 Top-Level Box Structure Design Development

The Top-Level Box Structure Design is developed by the chief designer from the Top-Level Box Structure Specification. The format of the Top-Level Box Structure Design follows the Specification and Development Style Guide, and it is developed with reference to Common Service Documentation. New common service functions may be created or existing ones updated.

Top-Level Box Structure Design - the system architecture; the top-level clear box in the box structure Usage Hierarchy.

Common Service Documentation - the black box descriptions of reusable box services.

2.1.4 Top-Level Box Structure Design Review

The Top-Level Box Structure Design is reviewed by the development team using the Design Review Protocol. If changes in the design are necessary or desirable, another iteration of Design Development and Review occurs.

2.1.5 Top-Level Usage Specification Development

The Usage Specification is developed by the certification team using Customer Requirements. Clarification of requirements, if needed, is obtained through customer interaction.

Usage Specification - a description of the expected users of the software, the expected uses of the software, and the expected system environment.

2.1.6 Top-Level Usage Specification Review

The Usage Specification is reviewed by the full team and representative users employing the Usage Model Review Protocol with reference to the Customer Requirements. If changes in the Usage Specification are necessary or desirable, another iteration of Usage Specification Development and Review occurs.

2.2 Increment Planning

2.2.1 Team Capability and Performance Evaluation

The capability of individual team members is assessed through a review of educational and professional backgrounds. The capability of the team is assessed through a review of Project Records reflecting the team's performance on similar projects. Training needs are identified.

2.2.2 Increment Plan Development and Maintenance

The Incremental Development Plan is prepared with reference to the Project Schedule, the Top-Level Box Structure Specification, and the results of Team Capability and Performance Evaluation. It is formulated after consideration of the clarity of requirements, usage probability of user functions, reliability requirements for subsystems, coordination with related development schedules, dependencies between functions, complexity, reuse, or other factors that pose risks to the project. In the first increment, the Incremental Development Plan is based on the initial schedule, specification, and team evaluation. In each subsequent increment, the team reviews the continued viability of the plan in light of the current schedule, specification, and team evaluation. The plan is revised if necessary.

Incremental Development Plan - a plan that specifies the number of increments into which a Cleanroom Project will be divided, and identifies the functions that will be implemented in each increment. The Incremental Development Plan is a component of the Project Schedule.

2.3 Box Structure System Decomposition

2.3.1 Individual Specification and Design

Development team members use the Top-Level Box Structure Specification and the work in previous increments to create system parts as assigned by the chief designer.

Box Structure Files - the complete software specification and design represented in black box, state box, and clear box forms.

2.3.2 Team Specification and Design Review

The development team reviews box structure parts drafted by individuals. If changes in a box are necessary or desirable, another iteration of Individual Specification and Design and Team Specification and Design Review occurs. If the changes have system-wide implications, a new top-down review is done. If changes are not needed and the system is complete, development team members proceed with team verification.

2.3.3 Team Verification

The Verification Protocol is used to verify correctness of either the entire design or the most recent level of decomposition. The entire design should be verified if questions about system integrity arise, development team members are found to be less familiar with the system than they should be, or changes in the previous version have system-wide implications. A Verification Fault Form and the Fault Classification Scheme are used to document each fault found during verification. If changes to the Box Structure Files are necessary or desirable, another iteration of Box Structure System Decomposition occurs.

Usage Hierarchy - the system architecture, i.e., the relationship among Box Structure Files.

Verification Meeting Records - completed Verification Meeting Forms which contain information relating to all verification meetings for the project.

Verification Fault Records - completed Verification Fault Forms which contain information relating to all faults found during verification.

Verified Box Structure Files

2.4 Statistical Test Planning

2.4.1 Usage Model Development

The certification team develops the Usage Model using Customer Requirements, the Top-Level Box Structure Specification, and the Usage Specification. The structure of the usage model is validated through automated analysis. If the structure is invalid, the model is revised until the structure is valid.

Usage Model - The Usage Model is a Markov model representing the software as a stochastic process. It represents the usage states of the software and the probabilities of transitions between usage states. When the software is to be certified for normal operational use, frequencies or probabilities of each event in the usage model structure are assigned based on expected use of the software. When the customer requires certification for safety-critical use, frequencies or probabilities also reflect the criticality of system functions.

2.4.2 Usage Model Review and Analysis

The usage model structure is verified against the Top-Level Box Structure Specification. The initial probability distribution is verified against any information available concerning intended usage of the software, especially the Usage Specification. A Markov Usage Model Analysis is automatically generated and reviewed to discern the viability of the model as a basis for testing. A few random test cases are also generated for review. If changes in the model are necessary or desirable, another iteration of Usage Model Development and Usage Model Review and Analysis occurs. A hypothetical testing scenario (e.g., distribution of failures over test cases) is developed, and number of test cases that must be run to meet the target level of reliability is determined. If this number is too high to be practical for the defined test environment, a cost-benefit analysis is done to determine how the scope of the test environment should be revised.

Usage Model Analysis - a Markov analysis of a usage model, providing analytical results for

- the expected length of a test case in states,
- the expected minimum number of test cases until all states in the usage model have been visited at least once,
- the percentage of time that will be spent in each state in the long run, and
- the expected number of test cases that must be run before a given state is visited.

2.4.3 Test Planning

The certification team prepares the Test Script using the Usage Specification, the Top-Level Box Structure Specification, and the Usage Model. If confirmation of correct system function requires an independent authority, such an "oracle" is identified and acquired. Test cases (a complete usage scenario in a sequence of user inputs) are randomly generated from the model. The test cases are the basis for the Test Script. Instructions to testers on test execution and validation are added to the Test Script. The certification team meets at the conclusion of Test Planning to ensure a common understanding of the Test Script, Testing Policies, relevant Data Collection Forms, and any other references or systems that will be used as an "oracle" during testing.

Test Script - all plans needed by testers in the application of randomly generated test cases during certification testing. The Test Script consists of test cases and instructions to testers for their execution and validation.

2.5 Increment Certification

2.5.1 Compilation

A certification team member compiles the Box Structure Files. If the compilation was unsuccessful, faults are found and fixed in the concurrent process of Engineering Changes. Compilation is repeated until the system is compiled without errors or warnings. Compilation sessions are documented.

Compilation Session Records - completed Compilation Session Forms, which contain information about all compilation sessions for the current project.

2.5.2 Testing

The certification team executes the Test Script and uses Testing Policies to compare observed performance with expected performance according to the Top-Level Box Structure Specification. All "tester's choice" decisions and unexpected outcomes are noted on the Test Script. A Testing Failure Form is used to document each failure. The completed forms become the Testing Failure Records. Failures are examined and faults are fixed in the concurrent process of Engineering Changes.

Annotated Test Script

Testing Failure Records - completed Testing Failure Forms which contain information relating to all failures found during testing.

2.5.3 Reliability Estimation

A Testing Failure Summary Form is used to summarize relevant data from Testing Failure Records. Automated tools are used to produce Reliability Reports.

When the reliability goal is reached, the system is "certified" as fit for use in the manner represented in the usage model. If the reliability goal is not reached on the expected schedule, testing stops and the increment is returned to the development team for reengineering.

Testing Failure Summary - a summary of the information contained in the Testing Failure Records, i.e., all of the information relating to failures found during testing.

Reliability Reports -

- estimates of reliability and mean time between failure (MTBF) that may be considered as stopping criteria for testing a version on which failures have occurred, and
- a measure of discrimination that may be considered as a stopping criterion for testing the final version.

2.5.4 Engineering Change Design

The Usage Hierarchy and relevant Box Structure Files are reviewed with reference to the Development References to identify the fault which caused the observed failure. If an engineering change is required to fix a fault which resulted in a failure during Testing, development team members also review Testing Failure Records and the Annotated Test Script and consult certification team members as needed to understand the observed failure. If an engineering change is required to fix failures during Compilation, development team members also review the output of the compiler to determine the number and nature of the faults discovered. A Compilation Fault Form is used to document each fault that is found.

For each box examined, the investigation proceeds as follows:

- the black box is reviewed to determine whether the problem is in the specification,
- the state box is reviewed to determine whether the problem is in the definition of state data or the specification for updating state data, and
- the clear box is reviewed to determine whether the problem is in the design.

If the fault cannot be identified through this review, a prototyping experiment may be conducted within the guidelines given in Development Policies. Once the fault is fully understood, it is classified using the Fault Classification Scheme and the engineering change is designed. When the engineering change has been designed, the relevant parts of the Box Structure Files are changed. An Engineering Change Form is used to document each change.

Compilation Fault Records - completed Compilation Fault Forms which contain all information relating to the faults found during Compilation.

Engineering Change Records - completed Engineering Change Record Forms and the annotated box files to which changes were made.

Updated Box Structure Files

2.5.5 Engineering Change Verification

Development team members use the Verification Protocol and Engineering Change Records to verify correctness of an engineering change. Whether the engineering change was made correctly is noted in the Engineering Change Records. If it was not, a Verification Fault Form and the Fault Classification Scheme are used to document each fault.

Verified Box Structure Files

2.6 Increment Conclusion

2.6.1 Increment Report Preparation

The project manager coordinates the preparation of the Increment Report at the end of each increment. The chief designer coordinates the completion of Development Records, and the chief certification engineer coordinates the completion of Certification Records. The project manager coordinates the completion of all other Project Records.

Increment Report - development records, certification records, and lessons learned in an increment.

2.6.2 Process Review and Improvement

After each increment the team reviews the Verification Records, Compilation Records, and Testing Records to identify opportunities for process improvement. The Project Records are reviewed as needed to understand common or systematic errors. Process References are updated to reflect the team consensus on process improvement. Process Control Standards may be updated if new information enables the setting of new standards or revising of existing standards.

Updated Process Control Standards

Updated Process References

Reference

"Cleanroom Process Description." Software Quality Research Laboratory, Department of Computer Science, University of Tennessee, Knoxville, Tn., June 1995.

IV. OO PROCESSES AND CLEANROOM EXTENSIONS

BOOCH PROCESS - OVERVIEW

In the Booch process, "software growing" occurs through the iterative and opportunistic interplay of macro and micro processes in "round-trip Gestalt design."

Macro Process

The macro process is the controlling framework for the micro process. It dictates a number of measurable products and activities that permit the development team to meaningfully assess risk and make early corrections to the micro process, so as to better focus the team's analysis and design activities. The macro process is primarily the concern of the development team's technical management, focusing on technical progress, risk mitigation, and architectural vision. (pp. 248-249)

Micro Process

The micro process represents the daily activities of the software developers. It is driven by the stream of scenarios and architectural products that emerge from and that are successively refined by the macro process. It offers guidance in making tactical decisions, and offers a framework for evolving the architecture and exploring alternative designs. In the micro process, the traditional phases of analysis and design are intentionally blurred, and the process is under opportunistic control. (pp. 234-235)

Booch and Cleanroom

In Cleanroom, "software growing" occurs through incremental and formal development and certification activities under statistical process control.

On the surface, the Booch and Cleanroom approaches seem compatible because of their explicit process structure for "software growing." Both use cycles of technical activity within a larger umbrella of process control. In Booch, the technical cycle is the micro process; in Cleanroom, it is specification, design, verification, and certification. In Booch, the larger umbrella of control is the macro process; in Cleanroom, it is the incremental development plan.

The superficial similarity between the approaches is misleading, however. Booch emphasizes evolution; the requirements for the system evolve along with the architecture. In Cleanroom, a commitment to an architectural design is made at the outset, and every succeeding design decision is intended to preserve the integrity of the design. The Booch process, under opportunistic control, regards a system as an evolving "Gestalt" (whole). The Cleanroom process, under statistical process control, regards a system as an unfolding mathematical function.

Although the table that follows identifies specific opportunities for Cleanroom extensions to the Booch process, the two processes are philosophically at odds. Cleanroom engineering precision is, for Booch, overengineering.

BOOCH - GLOSSARY

behavior - how an object act or reacts, in terms of its state changes and message passing; The outwardly visible and testable activity of an object.

class - a set of objects that share a common structure and a common behavior. The terms *class* and *type* are usually (but not always) interchangeable; a class is a slightly different concept than a type, in that it emphasizes the classification of structure and behavior.

function - an input/output mapping resulting from some object's behavior.

function point - in the context of requirements analysis, a single, outwardly visible and testable activity.

message - an operation that one object performs upon another. The terms *message*, *method*, and *operation* are usually interchangeable.

method - an operation upon an object, defined as part of the declaration of the class; all methods are operations, but not all operations are methods. The terms *message*, *method* and *operation* are usually interchangeable.

object - something that you can do things to. An object has state, behavior, and identity. The structure and behavior of similar objects are defined in their common class. The terms *instance* and *object* are interchangeable.

protocol - the ways in which an object can act and react, constituting the entire static and dynamic outside view of the object; the protocol of an object defines the envelope of the object's allowable behavior.

state - the cumulative results of the behavior of an object; one of the possible conditions in which an object may exist, characterized by definite quantities that are distinct from other quantities; at any given point in time, the state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

round-trip gestalt design - a style of design that emphasizes the incremental and iterative development of a system, through refinement of different yet consistent logical and physical views of the system as a whole; the process of object-oriented design is guided by the concepts of round-trip gestalt design; round-trip gestalt design is a recognition of the fact that the big picture of a design affects its details, and that details often affect the big picture.

Reference

Booch, G. Object Oriented Analysis and Design, 2nd ed. Benjamin Cummings, 1994. Glossary, pp. 513-519.

BOOCH - PHASES

Figure 1 illustrates the Booch Process as described in the documentation baseline for the method.

PHASES OF THE MACRO PROCESS

1. Conceptualization

Conceptualization is concerned with establishing the core requirements of the system. Its purpose is to establish the vision for a new application idea and validate associated assumptions. Prototypes are the primary products of Conceptualization. There is nothing inherently object-oriented about Conceptualization. (pp. 250-251)

2. Analysis

The purpose of Analysis is to provide a model of the system's behavior. Analysis focuses on behavior, not form; it yields a statement of what the system does, not how it does it. The primary activities in Analysis are domain analysis and scenario planning. A complete analysis is neither expected nor desired; it is sufficient that only primary and some secondary behaviors be considered. (pp. 252-255)

3. Design

The purpose of Design is to create an architecture for the evolving implementation, and to establish the common tactical policies that must be used by disparate elements of the system. The primary activities in Design are architectural planning, tactical design, and release planning. (pp. 255-257)

4. Evolution

The purpose of Evolution is to grow and change the implementation through successive refinement of the system's architecture, ultimately leading to the production system. The requirements evolve along with the architecture. The primary product of Evolution is a stream of executable releases representing successive refinements to the initial architectural release. The primary activities in Evolution are application of the micro process and change management. The successful completion of the Evolution occurs when the functionality and quality of the releases are sufficient to ship the product. (pp. 257-263)

5. Maintenance

Maintenance is the activity of managing post-delivery evolution. This phase is mainly a continuation of Evolution, except that architectural evolution is less of an issue. The products and activities are similar to those of the previous phase. Continued production releases and intermediate bug releases are produced. (pp. 263-264)

Phases of the Micro Process

Identify Classes and Objects

The purpose of identifying classes and objects is to establish the boundaries of the problem at hand, and to devise an object-oriented decomposition of the system under development.

- In analysis, this step is applied to constrain the problem and decide what is and is not of interest.
- In design, this step is applied to invent new abstractions that form elements of the solution.
- In implementation, this step is applied to invent lower-level abstractions that can be used to construct higher-level ones, and to discover commonality among existing abstractions that can be exploited to simplify the system's architecture. (pp. 235-236)

Identify Semantics of Classes and Objects

The purpose of identifying the semantics of classes and objects is to establish the behavior and attributes of each abstraction identified in the previous phase. Candidate abstractions are refined through distribution of responsibilities.

- In analysis, this step is applied to allocate responsibilities for different system behaviors.
- In design, this step is applied to achieve a clear separation of concerns among the parts of the solution.
- In implementation, the free-form description of roles and responsibilities is transformed to a concrete protocol for each abstraction. (p. 238)

Identify Relationships among Classes and Objects

The purpose of identifying the relationships among classes and objects is to solidify the boundaries of and recognize the collaborators with each abstraction identified earlier in the micro process. This activity formalizes the conceptual as well as physical separations of concern among abstractions.

- In analysis, this step is applied to specify the associations among classes and objects (including inheritance and aggregation relationships).
- In design, this step is applied to specify the collaborations that form the mechanisms of the architecture, as well as the higher-level clustering of classes into categories and modules into subsystems.
- In implementation, this step is applied to refine relationships such as association into more implementation-oriented relationships, including instantiation and use. (p. 242)

Implement Classes and Objects

Decisions about the representation of each abstraction and the mapping of these representations to the physical model drive the products from this step. The primary activity associated with this step is the selection of the structures and algorithms that provide the semantics of the abstractions. Whereas the first three phases of the micro process focus on the outside view of abstractions, this step focuses on the inside view.

- In analysis, the purpose of implementing classes and objects is to provide a refinement of existing abstractions sufficient to unveil new classes and objects at the next level of abstraction.
- In design, this activity is applied to create tangible representations of abstractions in support of the successive refinement of the executable releases. (pp. 246-248)

References

Booch, G. Object Oriented Analysis and Design, 2nd ed. Benjamin Cummings, 1994, pp. 513-519.

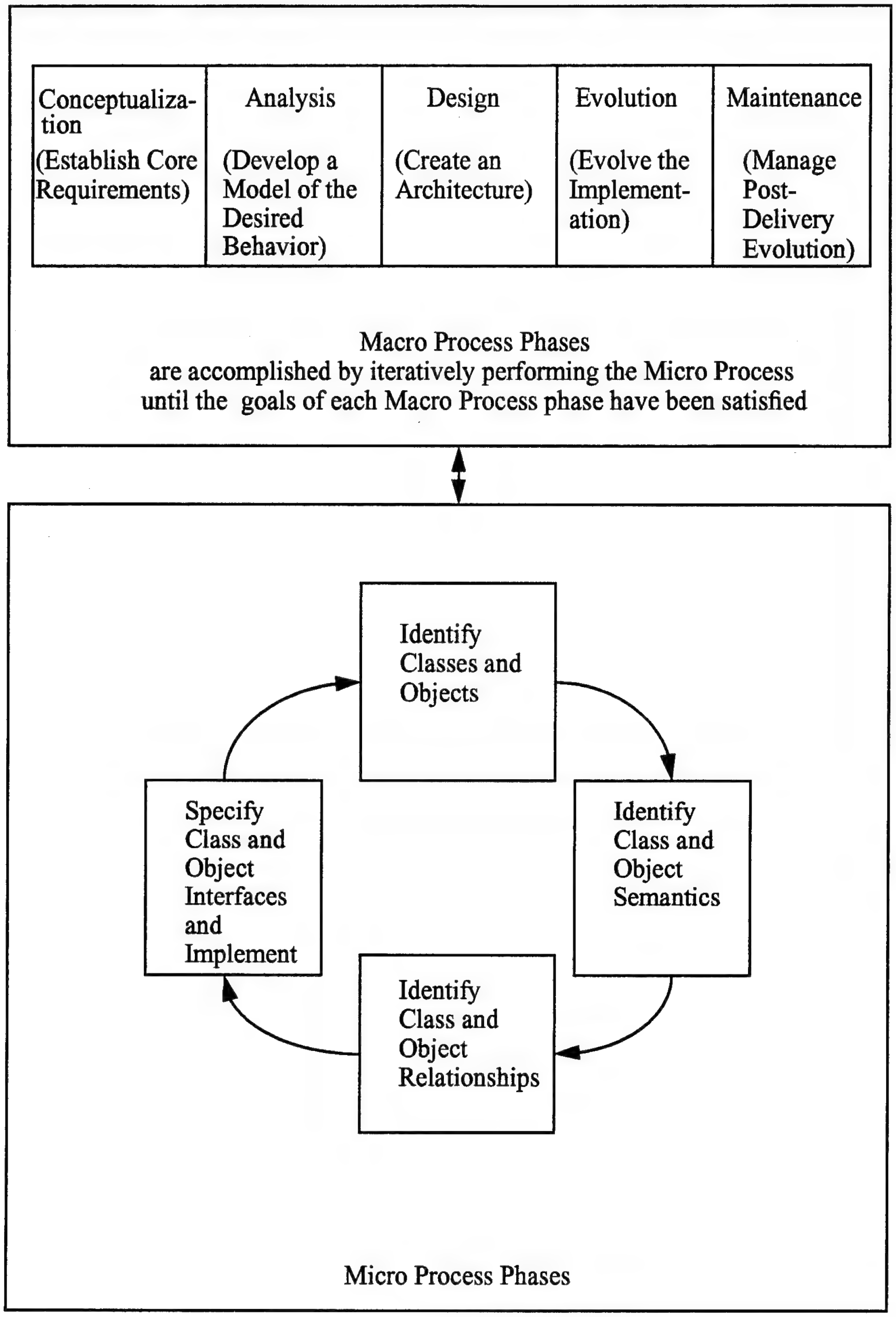

Conceptualiza-tion (Establish Core Requirements)
Analysis (Develop a Model of the Desired Behavior)
Design (Create an Architecture)
Evolution (Evolve the Implement-ation)
Maintenance (Manage Post-Delivery Evolution)
Macro Process Phases are accomplished by iteratively performing the Micro Process until the goals of each Macro Process phase have been satisfied
Identify Classes and Objects
Identify Class and Object Semantics
Identify Class and Object Relationships
Specify Class and Object Interfaces and Implement
Micro Process Phases

Figure 1: The Booch Process

BOOCH - ACTIVITIES AND WORK PRODUCTS

1. Conceptualization

([2], pp. 80-86)

Executable Prototype - software prototype developed to demonstrate key concepts of a proposed system to stakeholders and decision makers.

Risk Assessment - description of areas of technical and non-technical risk that may impact the project and the design process.

Vision of Project's Requirements - document or prototype that illustrates the mission of a proposed system and its key requirements.

Micro Process Work Products (see below).

2. Analysis

2.1 Domain Analysis

Domain analysis involves identification of classes and objects that are common in the problem domain. ([1], p. 157, p. 253)

Domain Model - describes the classes of a problem space and the roles and responsibilities for each class.

Micro Process Work Products (see below).

2.2 Scenario Planning

Scenario planning involves identification of behaviors that are central to the application's purpose---primary function points, and secondary scenarios that illustrate behavior under exceptional conditions. ([2], p. 88)

System Context Description - description of the boundaries of the proposed system using a context diagram.

Scenarios - descriptions of all fundamental system behaviors.

Revised Risk Assessment - identification of newly-discovered areas of technical and non-technical risk that may impact the design process.

Micro Process Work Products (see below).

3. Design

([2], p. 108)

3.1 Architectural Planning

Architectural planning involves devising the layers and partitions of the overall system. It encompasses a logical decomposition, representing a clustering of classes, as well as a physical decomposition, representing a clustering of modules and the allocation of functions to different processors. ([1], p. 256)

Architecture Description - clustering of function points from the products of analysis, and allocation of the function points to layers and partitions of the architecture; architecture is validated in an executable release.

Executable and Baselined Architecture - real application developed to implement all key architectural interfaces, and implement portions of a system to investigate performance issues or to examine system behavior defined in selected scenarios.

Micro Process Work Products (see below).

3.2 Tactical Design

Tactical design involves making decisions about a myriad of common policies. Some policies may be domain independent, such as memory management and error handling, and some domain-specific, such as control policies in real-time systems or database management in information systems. ([1], p. 256)

Description of Common Tactical Policies - scenarios describing the semantics of each tactical design policy; validated in an executable prototype.

Micro Process Work Products (see below).

3.3 Release Planning

Release planning involves identification of a controlled series of architectural releases, each growing in its functionality, ultimately encompassing the requirements of the complete production system. ([1], p. 255-257)

Release Plan - a formal development plan which identifies the stream of architectural releases, team tasks, and risk assessments.

Test Criteria - test plan, and test script derived from usage scenarios. ([2], p. 115)

Revised Risk Assessment.

Micro Process Work Products (see below).

4. Evolution

4.1 Application of the Micro Process

Preparation of a stream of executable releases and special behavioral prototypes to investigate and define issues of system and software behavior. ([2], pp. 108-129, pp. 158-184)

Executable Releases - executable versions of the system used for evaluation and testing during development.

Behavioral Prototypes - special versions of the system or special programs prepared to work out issues regarding how the system, or a class of the system should behave.

System and User Documentation.

Quality Assurance Results.

Micro Process Work Products (see below).

4.2 Release Assessment and Change Management

Release assessment and change management exist in recognition of the incremental and iterative nature of object-oriented systems. They involve adding, changing, or reorganizing classes or the class structure. ([2], pp. 136-143)

Micro Process Work Products (see below).

5. Maintenance

Not in the scope of this study.

MICRO PROCESS WORK PRODUCTS

Identify Classes and Objects

Data Dictionary - a central repository for the abstractions of the system.

Identify Semantics of Classes and Objects

([2], p.167)

Key Abstraction Roles and Responsibilities Specifications - identifies the abstractions for a system, the functional responsibilities assigned to each abstraction, and the objects/classes with which the abstraction must communicate.

Executable Software.

Diagrams and charts that establish the meaning of each abstraction, which include:

Object Diagrams - shows the existence of objects and their relationships in the logical view of a system.

State Transition Diagrams - shows the state space of a given class, the events that cause a transition from one state to another, and the actions that result from a state change.

Interaction Diagrams - a restructured representation of the essential elements of an object diagram; used to trace the execution of a scenario in the same context as an object diagram; represents the passing of messages in relative order. [[1], p. 217)

Updated Data Dictionary.

Identify Relationships Among Classes and Objects

([2], pp.174-176)

Key Abstraction Relationship Specifications - identifies the relationships among abstractions for a system.

Executable Software.

Updated products of previous phases.

Diagrams or charts that establish the meaning of each relationship as well as larger collaborations, such as:

> *CRC (Class/Responsibilities/Collaborators) Cards* - describes the key abstractions and mechanisms of a system, and the collaboration responsibilities of each class. ([2], p.174)
>
> *Module Diagrams* - shows the allocation of classes and objects to modules in the physical view of a system. ([1], p.242)

Implement Classes and Objects

Executable Software.

Updated products of previous phases.

This code may be supported by: ([1], p.247-248)

> *Process Diagrams* - shows the allocation of processes to processors in the physical view of a system.
>
> *Pseudocode.*

References

[1] Booch, G. Object-Oriented Analysis and Design with Applications, 2nd edition. Benjamin-Cummings, 1994.

[2] Booch, G. Object Solutions: Managing the Object Oriented Project, Addison Wesley, 1996.

CLEANROOM EXTENSIONS TO BOOCH PROCESS

The Booch process is given in the left column as the "base" process, and the Cleanroom process in the right column is mapped to the Booch process. Comments that straddle the columns are recommended Cleanroom extensions to the Booch process. Cleanroom extensions are given immediately after the Booch process part to which they apply.

BOOCH PROCESS	CLEANROOM PROCESS
	1. Project Management *Process References* *Project Records* **1.2 Process Control** *Process Control Standards* **1.3 Schedule Development and Maintenance** *Project Schedule* **1.4 Training** **1.5 Tasking** *Tasking Records*
Cleanroom extension: Represent the activities and work products in the Booch process in a manner that can be used as a "template" for process definition and project management. Rationale: A more explicit representation of processes and work products might better define the interplay of the Booch Macro and Micro Processes. Just as the Booch Data Dictionary serves as a common technical reference for team members, Cleanroom work products such as Process References, Process Control Standards, and a Project Schedule serve as common management references.	
1. Conceptualization *Executable Prototype* *Risk Assessment*	**2.2 Increment Planning** **2.2.1 Team Capability and Performance Evaluation** **2.2.2 Increment Plan Development and Maintenance** *Incremental Development Plan*

Booch Process	**Cleanroom Process**
Cleanroom extension: Prepare an initial incremental development plan to embody the results of Booch Risk Assessment. Rationale: An explicit purpose of Cleanroom Increment Planning is to assess and manage risk. The Incremental Development Plan can be a work product that is, in effect, the action plan for managing risk.	
Vision of Project's Requirements *Micro Process Work Products*	**1.1 Customer Interaction** *Customer Requirements*
2. Analysis **2.1 Domain Analysis** *Domain Model* **2.2 Scenario Planning** *System Context Description* *Scenarios*	**2.1.5 Top-Level Usage Specification Development** *Usage Specification* **2.1.6 Top-Level Usage Specification Review**
Revised Risk Assessment *Micro Process Work Products*	**2.2.2 Increment Plan Development and Maintenance** *Incremental Development Plan*
	2.1.1 Top-Level Box Structure Specification Development *Top-Level Box Structure Specification (black box)* **2.1.2 Top-Level Box Structure Specification Review**

BOOCH PROCESS	CLEANROOM PROCESS
Cleanroom extension: Prepare a black box specification for the system. Rationale: The top-level black box specification of external behavior identifies all user-generated stimuli and the corresponding system responses in all circumstances of use. In the preceding Booch analysis step, the fundamental system behaviors were identified. A subsequent formal black box system specification would eliminate remaining ambiguity by extending the set of defined behaviors from "fundamental behaviors" to "all behaviors." The system specification provides early conceptual integrity; it is a basis for understanding system parts in the context of the whole.	
3. Design **3.1 Architectural Planning** *Architecture Description* *Executable and Baselined Architecture* *Micro Process Work Products*	**2.1.1 Top-Level Box Structure Specification Development** *Top-Level Box Structure Specification (state box)* **2.1.2 Top-Level Box Structure Specification Review** **2.1.3 Top-Level Box Structure Design Development** *Top-Level Box Structure Design (clear box)* **2.1.4 Top-Level Box Structure Design Review**

BOOCH PROCESS	CLEANROOM PROCESS
Cleanroom extension: Prepare a state box specification for the system. Rationale: In the Booch design phase, the purpose of the first step of the micro process is to invent new abstractions that form elements of the solution. The derivation of the top-level state box from the top-level black box serves exactly this purpose, while preserving the conceptual integrity of the black box system specification. Stimulus history that is needed to determine correct system responses is identified, and state data objects are invented to encapsulate the history. The black box specification: Black box: (current stimulus, stimulus history) -> (response) is rewritten as a state box specification: State box: (stimulus, old state) -> (response, new state). Cleanroom extension: Examine the state box for transaction closure. Rationale: The Cleanroom principle of transaction closure is a check to ensure that an object's state data are sufficient to support its transactions, and the stimuli are sufficient to support all state data. Given transaction closure, the system-level specification of external behavior and internal state is an object now ready for the logical and physical decomposition that takes place in Booch Architectural Planning. Cleanroom extension: Ensure referential transparency by preparing a black box specification for each subsystem. Rationale: The Cleanroom principle of referential transparency requires clearly defined interfaces between an object (in this case, a subsystem) and its user(s). Once an object's function (i.e., the mapping of its stimulus set to its response set) is defined, its internal design may take place without further reference to other parts of the system.	
3.2 Tactical Design *Description of Common Tactical Policies* *Micro Process Work Products*	*Common Service Documentation*
3.3 Release Planning *Release Plan* *Revised Risk Assessment*	**2.2.2 Increment Plan Development and Maintenance** *Incremental Development Plan*
Cleanroom extension: Update the incremental development plan (that was created as an action plan for the initial Booch Risk Assessment) as the Booch Release Plan and Revised Risk Assessment. Rationale: The incremental development plan defines the schedule and content for implementation of function. Decisions about implementation priorities are largely based on risk assessment.	

BOOCH PROCESS	CLEANROOM PROCESS
	2.4 Statistical Test Planning
	2.4.1 Usage Model Development *Usage Model*
	2.4.2 Usage Model Review and Analysis *Usage Model Analysis*
Test Criteria	**2.4.3 Test Planning** *Test Script*
Micro Process Work Products	
Cleanroom extension: Develop a plan for statistical usage testing. Rationale: The Booch method notes that testing is tied to the scenarios developed early in the life cycle. The Cleanroom approach to testing supports this idea in a formal way. A Cleanroom statistical test plan, which is based on a Markov model of customer usage, can be developed as soon as the external system specification is in hand. The Markov usage analysis that occurs during test planning enables fine-grained decisions about test goals, resources, and schedule.	

BOOCH PROCESS	CLEANROOM PROCESS
4. Evolution	**2.3 Box Structure System Decomposition**
4.1 Application of the Micro Process *Executable Releases* *Behavioral Prototypes* *Micro Process Work Products*	**2.3.1 Individual Specification and Design** *Box Structure Files*
	2.3.2 Team Specification and Design Review
System and User Documentation	**2.3.3 Team Verification** *Usage Hierarchy*
Quality Assurance Results	**2.3.3 Team Verification** *Verification Meeting Records* *Verification Fault Records* *Verified Box Structure Files*
	2.5.1 Compilation *Compilation Session Records*
	2.5.2 Testing *Annotated Test Script* *Testing Failure Records*
	2.5.3 Reliability Estimation *Testing Failure Summary* *Reliability Reports*
4.2 Release Assessment and Change Management	**2.5.4 Engineering Change Design** *Compilation Fault Records* *Engineering Change Records* *Updated Box Structure Files*
Micro Process Work Products	**2.5.5 Engineering Change Verification** *Verified Box Structure Files*
Cleanroom extension: Certify the system using Cleanroom statistical certification. Rationale: The Booch method does not prescribe an approach to testing. Cleanroom certification testing is a customer usage-based approach that affords a disciplined framework for Booch Release Assessment and Change Management. Cleanroom certification provides engineering change control of the process and statistically valid measures of the product.	

BOOCH PROCESS	CLEANROOM PROCESS
	2.6 Increment Conclusion **2.6.1 Increment Report Preparation** *Increment Report* **2.6.2 Process Review and Improvement** *Updated Process Control Standards* *Updated Process References*
Cleanroom extension: Review performance in each increment for lessons learned. Rationale: A review of experience at the end of each increment of work should be included in the process structure mentioned in the first Cleanroom extension. The process review is the forum for group decisions about process improvement.	
5. Maintenance	(Regarded as a new increment. All increment activities apply.)
Micro Process Work Products: *Data Dictionary* *Key Abstraction Roles and Responsibilities Specifications* *- Object Diagrams* *- State Transition Diagrams* *- Interaction Diagrams* *Key Abstraction Relationship Specifications* *- CRC Cards* *- Module Diagrams* *Process Diagrams* *Pseudocode* *Executable Software*	

OBJECTORY PROCESS - OVERVIEW

The Objectory process is driven by the "use case" concept. A set of use cases that define all system behaviors is elaborated to a fully traceable design that is implemented through staged incremental development.

The Objectory process is the most well-defined of the OO processes included in this Guide, and probably has least to benefit from Cleanroom extensions. The Cleanroom approach to certification---statistical usage testing based on a Markov usage model---appears to have the most obvious added value for the Objectory process due to its extension of the utility of use cases.

Objectory and Cleanroom may have more in common than do Objectory and the other OO processes in this Guide. Both processes emphasize external specification, modeling of usage, design hierarchies, verification of and traceability between work products, and incremental development. The Objectory and Cleanroom processes would probably be the easiest and most productive blend of all the processes in this Guide.

OBJECTORY - GLOSSARY

acquaintance association - an association between two objects where one object knows about the existence of the other. ([7], pp. 38-40)

actor - represents something or someone external to a system that interacts with the system. ([7], p. 3). An actor can be a human user, external hardware, or another system.

alternative flow of events (of a use case) - is an alternative flow of events from the basic use case flow of events. Alternative courses describe odd and/or exceptional flows of events.

Analysis Model (Analysis-Object Model) - A model that describes the ideal structure for a system that is robust when subjected to changes and additions. It is described in terms of a structure of entity objects, interface objects, and control objects. ([7], p. 27). The interface object is used to model behavior that is surroundings-dependent (presentation). The entity object is used to model behavior that is surroundings-independent (persistent information). The control object is used to model behavior that is specific to a particular use case. [8]

attribute - an information element of an object; has a name, a cardinality, and a type; used for encapsulated information elements of an object. ([1], p. 512)

attribute-association - an association which links an attribute to an attribute type, where each attribute type and each attribute-association has a name. ([5], p.18)

basic flow of events (of a use case) - the flow of events of a use case that describe its normal and routine event flows. ([4], p. 10)

cardinality - the maximum number of acquaintance associations between two classes; how many instances may be associated by acquaintance. ([1], p. 178)

class - a definition, a template or a model to enable the creation of new objects. Objects of the same class have the same definition both for their operations and for their information structures. ([1], p. 51)

communication association - an association between two objects in which the objects exchange some information. ([7], p. 49)

consistsOf-association - an association from one analysis object to another analysis object, where the association describes the composition of one object in terms of the other. ([5], p.22)

containment hierarchy - a hierarchy of packages in which a parent package contains its child packages. Each package consists of a collection of collaborating objects. ([1], p. 181), [8].

Design Model (or Design-Object Model) - a documentation package and model of the system's implementation that serves as a blueprint for the structure and contents of the system's source code. The Design-Object Model represents the structure of the system, after it has been refined and adapted for the implementation environment, as an organization of subsystems and packages (or blocks) within subsystems. ([6], pp. 13-15)

design object - An object that represents an abstraction of objects in the system's implementation. ([4], p. 8). Normally, one design object is created for each analysis object identified. ([4], p. 47). *Design object* is synonymous with the OOSE term for *block*.

Development Case Description - a document that describes how the Objectory process has been tailored to support a project: the processes to follow, the models to build, the artifacts to be prepared to document the models, and how the models and artifacts should be reviewed. ([3], pp. 7-13)

Domain Object Model - A model that describes each object which has a direct counterpart in the application environment. This model describes what each object does and how it is used. This model also describes the behavior of each object, and its associations and relationships with other objects. ([1], p. 174; [2], p. 77; [4], pp. 97-111)

extends-association - an association between two analysis objects that indicates the associating object class may contribute in extending the behavior and information of the associated object's instances. Each extends-association has a condition. The extends-association is used to describe optional services of an object that may be realized only if selected conditions are met. ([5], pp. 40-41)

inheritance - if class B inherits class A, then both the operations and the information structure described in class A will become part of class B. ([1], p. 58)

inheritance-association - an association between two objects that defines the inheritance of one or more properties of the ancestor class by its descendents. ([5], pp. 42-43)

instance - an object created from a class. The class describes the (behavior and information) structure of the instance, while the current state of the instance is defined by the operations performed on the instance. ([1], p. 50)

message - see stimulus

object - an entity that encapsulates data and behavior, where the data is represented by an object's attributes and its behavior is represented by the object's operations. An object may only be communicated with through its interface. ([7], p. 11). An object is characterized by a number of operations and a state which remembers the effect of these operations. ([1], p. 44). An object represents both the class and all its instances. [8]

operation - an event of interest to an object or system, which causes the object or system to change internal state, and/or generate a response. ([1], pp. 44, 49)

Requirements Model - A model that describes 1) the functionality for the system, its interaction with system users, and its external interface descriptions ([1], p. 157), and 2) what each object does or how it is used, how it behaves, and what its associations are with other objects ([4], pp. 83-85). The focus of the requirements model is on the usage of the system as defined by the use cases. [8]

service package - the lowest level of a subsystem; an atomic managing unit. ([1], p. 512)

signal - see stimulus

stimulus - the event of one object communicating with another. In a programming context, the word "message" is often used instead but, in order to avoid the message semantics, we use the stimulus concept. Additionally, the word "stimulus" indicates that it stimulates some behavior to take action and does not necessarily include any message information. ([1], p. 47)

A stimulus can have different semantics, for example be either interprocess or intra-process communication. An intra-process stimulus is a normal call inside one process. We normally call this kind of stimulus a message. ... An interprocess stimulus is a stimulus sent between two processes. We usually call such a stimulus a signal. Signals may be either synchronous or asynchronous (the execution of the sender continues directly after the signal is sent). ([1], p. 221)

subscribesTo-association - an association between two analysis objects that indicates the subscribing object will be informed when a particular event has occurred in the associated object. Each subscribesTo-association has a condition. The condition defines the event that causes the subscriber to be notified. ([5], p. 30)

subsystem - a managing unit of objects and subsystems. ([1], p. 512)

user - anything which needs to exchange information with the system. ([1], p. 157)

use case - a behaviorally related sequence of transactions that represents a dialogue between the system and one or more actors. The collected use cases specify all ways of using the system. ([1], p. 159)

Use-Case Description - Analysis - A composition of descriptions and diagrams that show how the use cases' flow of events are performed by means of interacting analysis objects. ([5], pp. 154-157)

Use-Case Description - Design - A composition of descriptions and diagrams that show how the use-cases' flows of events are performed by means of communicating design objects. ([6], p. 119)

Use Case Model - There are three different flavors of the use case model: use cases for requirements analysis, use cases for robustness analysis, and use cases for design. A use case model for requirements analysis is a model that defines what should be performed by the system, what exists outside the system (actors), and the interaction between the system and the users of the system. ([7], p.14). The requirements analysis use case model should describe a use case's flow of events in terms a customer will understand. The use case model represents the packaging of all use cases for a system, and specifies all ways a system can be used. ([4], p. 71). A use case model for robustness analysis describes how the use cases' flow of events are performed by means of interacting objects both internal and external to the system. ([5], p. 155). A use case model for design describes how the use cases' flow of events are performed by means of communicating design objects. ([6], p. 117).

References

[1] Jacobson, Ivar, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach, revised 4th printing. Addison-Wesley Publishing Company, Reading, Massachusetts. 1993.

[2] Objectory AB. Objectory: Project Management, Version 3.6, 1995.

[3] Objectory AB. Objectory: Configuring your Process, Version 3.6, 1995.

[4] Objectory AB. Objectory: Requirements Analysis, Version 3.6, 1995.

[5] Objectory AB. Objectory: Robustness Analysis, Version 3.6, 1995.

[6] Objectory AB. Objectory: Design, Implementation and Testing, Version 3.6, 1995

[7] Objectory AB. Objectory: Introduction, Version 3.6, 1995.

[8] Aronsson, Johan. Personal Communication. January 31, 1996.

OBJECTORY - PHASES

The Objectory process spans the systems development life cycle. It is intended to be integrated with a project management process that is used to establish, plan and manage a systems development effort. The Objectory process supports the typical project management phases of Prestudy, Feasibility Study, Establishment, Execution, and Conclusion. The Objectory Requirements Analysis activity supports the Prestudy and Feasibility phases and the initial aspects of Project Execution. All other Objectory activities are performed during the Execution phase of a project. The work products of Requirements Analysis are typically only revised, as appropriate, during the Execution phase. ([2], pp. 7-9)

Figure 2 provides an illustration of the Objectory Process, as described in the documentation baseline for the method.

1. Project Planning

The aim of the Project Planning phase is to tailor the Objectory process for a specific project, and integrate the resulting tailored process with the organization's project management process. This phase is also concerned with planning the configuration management requirements necessary to support Objectory on a project, defining project roles and responsibilities, and analyzing the scope of the system and the scope of the work to be performed, as a precursor to project planning. The final aspect of this phase is to plan the execution portion of the project. ([2], pp. 1-46)

2. Analysis

The aim of the Analysis phase is to analyze, specify and define the system to be built. The models that are developed during analysis are fully application oriented, and no consideration is taken of the real implementation environment where the system is to be realized, for instance: the programming language, DBMSs, process distribution, or hardware configuration to be used. The purpose is to formulate the problem and build models able to solve the problem under ideal conditions. ([1], p. 148)

Analysis involves the activities of Requirements Analysis and Robustness Analysis, which results in an implementation-independent logical design.

3. Construction

3.1 Design

In Design, the analysis model is refined, formalized, and adapted to the implementation environment. The resulting design model accommodates, for example: performance requirements, real time requirements and concurrency, system software, properties of the programming language, and the database management system to be used. ([1], pp. 196-197)

Design involves the activities of implementation environment analysis and physical system design which results in a design object model. The design object model, in turn, serves as the blueprint for implementing the system.

3.2 Implementation

Implementation involves the actual coding of the system.

4. Testing

The focus of testing is verification that the system is being built correctly. The purpose of testing is to find faults. Several of the available types of testing (e.g., unit testing, integration testing, regression testing, etc.) should be used in combination. Testing activities are performed throughout development. ([1], pp. 307, 310, 332-3)

References

[1] Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

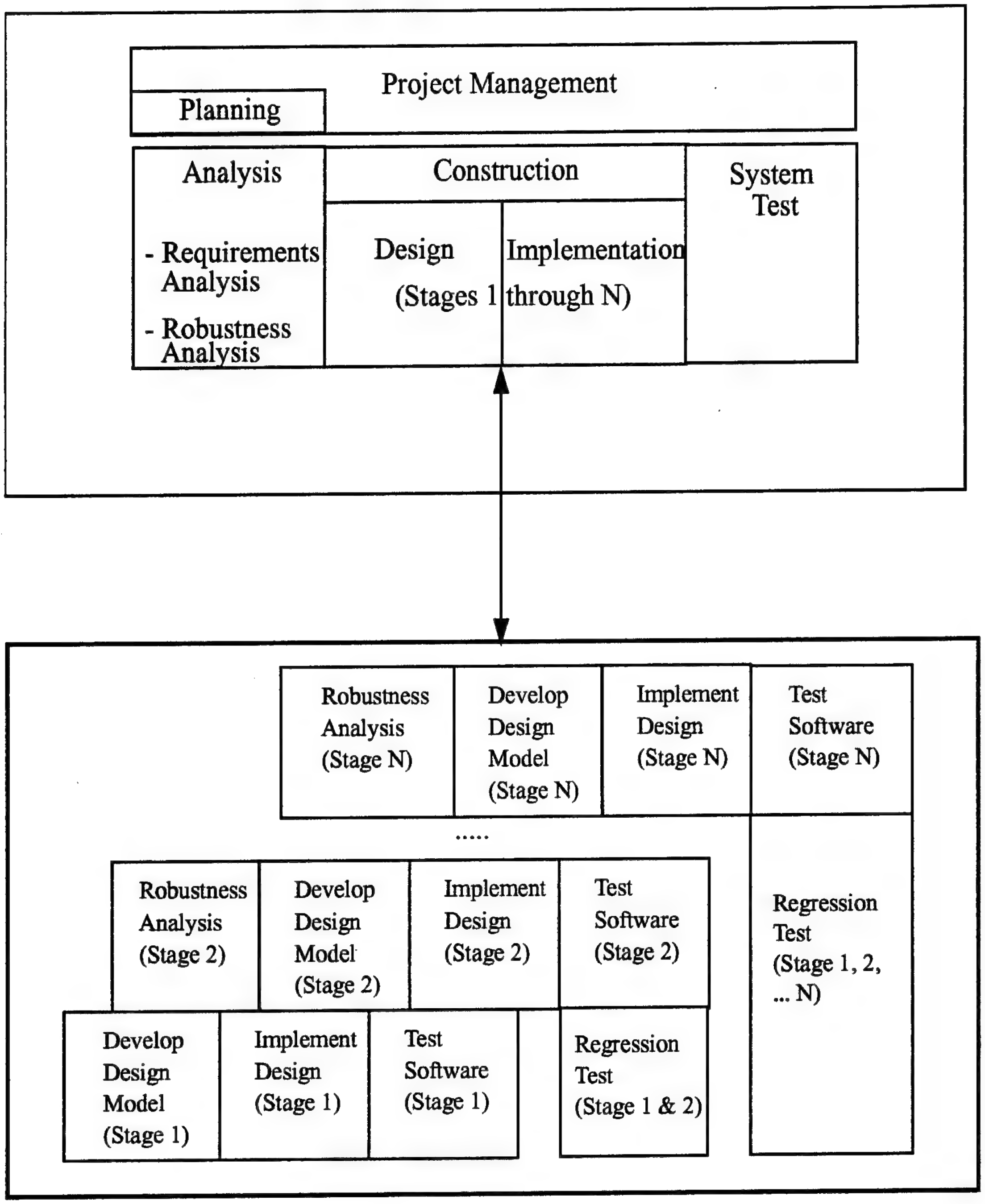
[2] Objectory AB. Objectory: Project Management, Version 3.6, 1995.

[3] Objectory AB. Objectory: Configuring your Process, Version 3.6, 1995.

[4] Objectory AB. Objectory: Requirements Analysis, Version 3.6, 1995.

[5] Objectory AB. Objectory: Robustness Analysis, Version 3.6, 1995.

[6] Objectory AB. Objectory: Design, Implementation and Testing, Version 3.6, 1995.


Project Management
Planning
Analysis
- Requirements Analysis
- Robustness Analysis
Construction
Design
Implementation
(Stages 1 through N)
System Test
Robustness Analysis (Stage N)
Develop Design Model (Stage N)
Implement Design (Stage N)
Test Software (Stage N)
.....
Robustness Analysis (Stage 2)
Develop Design Model (Stage 2)
Implement Design (Stage 2)
Test Software (Stage 2)
Regression Test (Stage 1, 2, ... N)
Develop Design Model (Stage 1)
Implement Design (Stage 1)
Test Software (Stage 1)
Regression Test (Stage 1 & 2)

Staged Incremental Development

Figure 2: The Objectory Process

OBJECTORY - ACTIVITIES AND WORK PRODUCTS

1. Project Planning

1.1 Objectory Process Configuration

The Objectory process is tailored to meet the needs of a systems development effort.

Development Case Description - a document that describes how Objectory has been tailored to support a project: the processes to follow, the models to build, the artifacts to be prepared to document the models, and how the models and artifacts should be reviewed. ([3], pp. 7-13)

1.2 Objectory/Project Management Integration

The Objectory process is integrated with the selected project management process. The project management process describes the activities of establishing, planning, estimating, tracking and controlling a systems development effort. This integration maps the activities and products of Objectory onto the phases, activities, and milestones of the project management process. ([2], pp. 7-9)

Tailored Project Management Process - a version of an organization's project management process, tailored for an Objectory systems development effort.

1.3 Objectory/Configuration Management Integration

The Objectory process is integrated with the selected configuration management process. The resulting integrated process will identify the key Objectory process artifacts that must be placed under configuration control, such as the Objectory models. This integration will address the version, configuration, and change control required for key Objectory products and their components. ([2], pp. 11-12)

Tailored Configuration Management Process - a version of an organization's configuration management process, tailored for an Objectory systems development effort.

1.4 Project Scoping

Each task identified in the "Development Case Description" is examined for its scope, level of ambition, and what it takes to perform it. Tasks are examined with respect to technical and environmental factors that affect the time it takes to perform them, as well as their level of complexity. The scope of the system will be addressed by examining the models to be produced, the documentation necessary for each model, and the level of ambition expected for models and associated documentation to be produced. Typical measures that can be used to support system scoping are the total number of use cases, use case size, and use case complexity. ([2], pp. 41-45)

No formal artifacts are identified for this activity.

1.5 Project Organization

Project technical staffing requirements are prepared using the Development Case Description and the recommendations made in the Objectory Project Management volume. ([2], pp. 33-40)

Project Technical Staffing Requirements - a description of the roles, responsibilities, and project organization necessary to support an Objectory systems development effort.

1.6 Project Planning

The systems development project is planned using the incremental development strategy (see 2.2.6), the Development Case Description and the results from project scoping. ([2], pp. 25-31)

Project Plan - a plan that addresses a project's cost and schedule requirements.

1.7 Development Case Installation

After the Development Case Description has been prepared and the project planned, it should be distributed to the development team, and training sessions should be held to explain its contents.

Trained Personnel - Personnel trained in the processes to be used to support the systems development effort.

1.8 Development Process Improvement

Measurements, results, and feedback are examined after each project to identify ways to improve the Development Case Description for future projects.

Development Case Improvement Recommendations - recommendations prepared from reviewing project measurements and results and from holding debriefing sessions.

2. Analysis

2.1 Requirements Analysis

The system to be built is analyzed, specified, and defined with respect to what the system will do and not how it will do it. In other words, the requirements analysis is conducted with as little reference to the implementation environment as possible. ([1], pp. 154, 156)

Requirements Model - (1) the Use Case Model, which describes the functionality for the system, interaction between system users, and external interface descriptions ([1], p. 157), and (2) the Domain Object Model, which describes each object's purpose, behavior, and associations with other objects ([4], pp. 83-85).

Activities 2.1.1 through 2.1.5 are required to prepare and validate the components of the Requirements Model.

2.1.1 User (Actor) Identification

All users of the system are identified and modeled as actors, where an actor is a user category or type. Note that a user is anything which needs to exchange information with the system. ([1], pp. 157-158)

Primary Actor Definitions - the list of users (person, external system, machine, etc.) who will interact directly with the system in their daily work, along with a description of each.

Secondary Actor Definitions - the list of users who will interact with the system in a supervisory or maintenance role.

2.1.2 Specification of Use Cases

All ways in which actors can interact with the system are specified as use cases. The sum of all use cases constitutes the system functionality. For each use case, basic courses and alternative courses are specified. ([1], p. 159)

Use-Case Model - artifacts that describe the use cases for a system. A use case is a behaviorally related sequence of transactions that represents a dialogue between the system and one or more actors. The collected use cases specify all ways to use the system. ([1], pp. 127-128) The use-case model is composed of the following artifacts:

- Use-Case-Model Survey - an overview of all use cases and actors (one per system)
- Use-Case Description: Requirements Analysis - complete documentation of each use case (one per use case)
  - Brief description (from the Use-Case-Model Survey)
  - Flow of events - flow of transactions between the system and its users (actors); includes a subsection called Alternative flows that documents exception processing (i.e., the flow of transactions different from the basic flow)
  - Associations - description of all uses-, extends-, and communication-associations
  - Special requirements - processing requirements such as performance and timing requirements; constraints on the system's functional requirements
  - User interface views - the flow of events in sketches or in the form of a prototype
  - Views - illustration of specific communication-associations with each actor
  - State-Transition Diagrams - describes the dynamic behavior of the use case, and, if necessary, its actors. ([4], pp.65-74)

2.1.3 Design Interfaces

Interface Description - a detailed description of all system interfaces, including man-machine interfaces and communication protocols. ([1], p. 166)

2.1.4 Identify Problem Domain Objects and Refine

Problem domain objects are objects about which the system should handle information. These objects form a glossary that can be used to formulate the functionality in the use cases. The resulting problem domain model provides a tool to communicate about the system. The domain objects are then tailored for the particular system. ([1], p. 167)

Domain-Object Model - a description of each object which has a direct counterpart in the application environment. This model describes what each object does and how it is used. This model also describes the behavior of each object, and its associations and relationships with other objects. ([1], p. 174; [4], pp. 97-111). The Domain-Object Model is composed of the following artifacts:

- Domain Object Model Survey - overview of all domain objects
- Catalog of Domain Objects - structure, attributes, associations, and operations of each object
- Use-Case Description: Domain Object Modeling - describes how the use case's flow of events are performed by means of communicating domain objects (one per use case)
- Domain-Object Description - complete description of each domain object (one per domain

object). ([4], pp.97-111)

2.1.5 Validate Requirements Analysis Products

The Use-Case Model and the Domain-Object Model produced during Requirements Analysis are reviewed relative to documented product validation criteria. ([4], pp. 62-64, pp. 92-95).

2.1.6 Review Requirements Analysis Activity Completion Conditions

Requirements Analysis phase activities are reviewed against completion criteria. ([4], pp. 59-60, pp. 90-91)

2.2 Analysis Model Development

Analysis model development is a "robustness analysis." Its principal product is the Analysis Model (or Analysis-Object Model). In Analysis Model Development, Use-Case Descriptions serve as the point of departure. ([5], p. 9)

Analysis Model (Analysis-Object Model) - describes the ideal structure for a system that is robust when subjected to changes and additions. It is described in terms of a structure of entity objects, interface objects, control objects. These objects represent 1) information (state), 2) behavior (changing state), and 3) presentation (responses). ([1], p. 174). The analysis model represents a machine-independent logical design that describes (1) how the flow of events is performed by communicating objects, (2) the interaction between objects inside the system, (3) the interaction between interface objects and external agents, and (4) the dynamic behavior of each object. ([5], pp. 154-157)

Activities 2.2.1 through 2.2.7 are required to prepare and validate the Analysis Model.

2.2.1 Identify Interface Objects

Interface objects are identified to describe the communication between external agents and internal objects. Associations and their cardinality are defined for each interface object. The interface object associations that may be formed with other interface objects are: acquaintance-associations, consistsOf-associations, extends-associations, inheritance-associations, and communication-associations. Further, interface objects may form communication-associations with actors, entity objects, and control objects.

The interface objects are placed in a containment hierarchy. Each use case is examined to see if parts of it can be allocated to interface objects. ([1], p. 176)

Interface Object - An object used to describe the communication between the system's surroundings and its inner workings. ([5], p. 3). Interface objects are directly dependent on the system environment. The interface object description includes external and internal interfaces, associations with other objects, and a description of the object's behavior.

2.2.2 Identify Entity Objects

Entity objects are used to hold and update information about some system phenomenon, such as an event, a person, or some real-life object. An entity object is not specific to one use case. The values of its attributes and associations are given by an actor. Entity objects also perform internal system tasks. Entity objects show the logical data structure for the system. ([5], pp. 14-15)

Associations and their cardinality are defined for each entity object. Entity object associations that may be formed with other entity objects are: acquaintance-associations, consistsOf-associations, extends-associations, inheritance-associations, subscribesTo-associations, and communication-associations. Further, entity objects may form communication-associations, subscribesTo-associations and acquaintance-associations with control objects and interface objects. ([5], p. 48)

Entity Objects - An object used to describe information in the system, and the behavior associated with it. ([5], p. 3). The entity object's description includes its behavior, attributes, type, operations, and its associations and their cardinality. ([5], p. 3)

2.2.3 Identify Control Objects

Control objects are defined to support courses (sequences) of events. Use cases from requirements analysis are evaluated to determine what behavior has not been allocated; this behavior is allocated to control objects. ([1], pp. 190-191, [5], pp. 154-157)

Associations and their cardinality are defined for each control object. Control object associations that may be formed with other control objects are: extends-associations, inheritance-associations, and communication-associations. Control objects may also form communication-associations with interface objects, and acquaintance-associations with entity objects.

Control Objects - objects created to control or direct communication among objects that support a specified use case.

Use-Case Description: Analysis - a composition of descriptions and diagrams that show how the use cases' flow of events are performed by means of interacting objects. It is composed of the following:

- a brief description of each use case
- a description of its flow of events
- a description of all objects that participate in the use case's flow of events
- special requirements for the use case that should be addressed during design
- "use-case analysis - interaction diagrams" which describe how the participating objects interact with the flow of events described by the use case
- "usage case analysis - state-transition diagrams" which describe the dynamic behavior of the use case and, as appropriate, its actors.

2.2.4 Subsystem Definition

The system is broken down into hierarchies of subsystems wherever possible. Analysis objects are organized into subsystems and packages within subsystems according to the functional relationships between the objects. ([5], p. 107). Subsystems may also be formed as a layered system architecture, where specific functionality is placed in the topmost layer and the general system functionality, e.g., common services, is placed in the lower layer. ([5], p. 110-111). Subsystems are defined to make a system resistant to functional changes and to make it possible to order the system's services independently of each other. The goal is to have any change in a specific service be limited so that it will affect only the objects within the package that offers the service. ([1], p. 196; [5], pp. 115-117)

Responsibilities for a use case may be distributed across subsystems, where the use case could be abstracted into a separate use case for each subsystem involved. These abstract use case descriptions would contain the analysis objects that affect the subsystems in question. ([5], p. 113)

Subsystems - groups of functionally-related objects organized into manageable development units. ([1], p. 174; [5]; pp. 107-109).

Analysis-Subsystem Description - a description of each subsystem and each package within a subsystem. Each subsystem/package description discusses the processing it supports and its dependencies on other subsystems and their packages.

Analysis-Service-Package Description - a description of each service package in terms of the processing it supports and its dependencies on other packages.

Analysis-Object Model: Interaction Diagrams - diagrams that describe the flow of events in terms of stimuli transmissions between packages, between objects, and between packages and objects allocated to subsystems. One diagram is prepared for each "Use Case Description - Analysis."

Analysis-Object Model: State-Transition Diagrams - diagrams that describes the dynamic behavior of each analysis object in terms of its state transitions. One diagram is prepared for each analysis object.

2.2.5 Analysis-Object Model Preparation

After all objects (interface, entity and control) have been identified and assigned behaviors from the use cases, and a system architecture has been defined in terms of subsystems and their supporting packages, the Analysis-Object Model artifacts are prepared.

Analysis-Object Model (defined in 2.2) - a package of the following artifacts:

- Analysis Survey - describes how the system is structured into packages, and identifies the analysis objects included within each package
- Traceability from the Domain-Object Model to the Analysis-Object Model
- Traceability from the Analysis-Object Model to the Domain-Object Model
- "Use-Case Description - Analysis"
- Analysis-Subsystem Descriptions
- Analysis-Service-Package Descriptions
- Entity-Object Descriptions
- Interface-Object Descriptions
- Control-Object Descriptions
- Attribute Types-Analysis - describes and lists the attribute types identified in the Analysis-Object model. This document can be useful in achieving homogenous usage of all attributes in the Analysis-Object Model.
- Interaction Diagrams
- State-Transition Diagrams
- Views - describes specific relationships between packages, objects, actors, use cases and

attribute types. Views also are prepared to illustrate specific associations. ([5], pp. 145-170)

2.2.6 Incremental Development Stage Planning

Using the subsystem definition, a staged incremental development plan is prepared. ([2], pp. 25-26)

Incremental Development Plan - plan that describes how packages and objects will be developed, tested and demonstrated in stages. Common service objects, user interface objects and critical application objects are typically developed in the first increment. Other packages and objects are staged according to their importance. ([2], pp. 125-129)

2.2.7 Review Analysis-Object Model Products

Before proceeding to the construction phase, analysis products are reviewed relative to their documented completion criteria. ([5], pp. 129-144)

2.2.8 Review Analysis Model Development Activity Completion Conditions

Analysis Model Development activities are reviewed against documented completion criteria. ([5], pp. 127-128)

3. Construction

3.1 Design

The purpose of Design Model development is to prepare an implementation-dependent design that shows how the system will be realized in the implementation phase. The Use Case Descriptions and the Analysis-Object Model are the inputs for preparing the Design Model. In this activity, specific design objects are defined from the analysis objects, where each design object represents an abstraction of the object's implementation. ([6], p. 13-15).

Design Model (or Design-Object Model) - a documentation package and model of the system's implementation that serves an blueprint for the structure and contents of the system's source code. ([6], pp. 13-15). The Design-Object Model documents the structure of the system, after it has been refined and adapted to the implementation environment. It is an organization of subsystems and packages (or design objects) within subsystems. The communications between subsystems and between packages within subsystems are illustrated using interaction diagrams. ([1], pp. 147, 205)

Activities 3.1.1 through 3.1.6 are required to prepare and validate the Design-Object Model.

3.1.1 Identify Implementation Environment

The characteristics of the implementation environment are identified. Additionally, the people and organization involved in development are considered at this point for individual competence and division of labor. Further, analysis objects and the implementation environment are analyzed to define design objects that support the implementation environment of the system. ([1], p. 208, [6], p. 32-41)

Target Environment Description - the program languages, components, existing products, performance requirements, memory limitations, people, and organization involved in development.

3.1.2 Define Interaction

All required package and design object interaction for each use case is defined in an interaction diagram. Interaction diagrams are the primary tool for preparing "Use-Case Descriptions - Design," which is the compilation of all interaction diagrams for the system and its subsystems. ([1], p. 215, [6], p. 117)

Design Object Model - Interaction Diagram - diagrams showing how a use case is executed by communicating packages and design objects. One Interaction Diagram is prepared per use case. ([1], p. 215, [6], p. 117)

Use-Case Description - Design - the package of Interaction Diagrams which show how the use-case's flows of events are performed by means of communicating design objects. ([1], p. 215, [6], p. 117)

3.1.3 Define Stimuli

Stimuli are identified and this information is added to the Interaction Diagrams. ([1], p. 218)

3.1.4 Design Subsystem / Package Architecture

The structure for the system is formed by organizing the analysis objects defined in the Analysis Object Model into packages according to functional relationships between objects. These packages, in turn, are organized into a hierarchy of subsystems which forms the architecture for the system. ([5], p. 107-108). The resulting system architecture must satisfy the use case description for analysis and include each object in the Analysis Object model. All analysis objects in the Analysis Object model must satisfy the flow of events for the system's use cases, and all behaviors of the system's use cases must be completely distributed to analysis objects. ([5], p. 60)

The interface is defined for each subsystem and package within a subsystem. The flow of events between each package and each subsystem is captured in an interaction diagram. ([5], p. 168). The behavior of each package and each subsystem is captured in a state transition graph. ([5], p. 173). The package structure for a subsystem is organized with regard to each package's operations and functionality, e.g. application-specific functions, general functions applicable to the application, and non-application specific services, such as an operating system, a window management system, a database management system, and a communication package. ([6], pp. 82-85)

Subsystem Architecture - architectural description of the components of the system in terms of its Subsystem and Package within Subsystem organization. Each subsystem in the system architecture is described in terms of its processing, its dependencies on other packages, and its contents. ([5], pp. 107-115)

Package - an architectural unit. A package may consist of a logical grouping of design objects. A package may also be composed of lower-level packages. ([1], p. 229)

Design Object Model - State Transition Diagram - a graph that depicts the behavior of each design object in the form of states and their allowable state transitions. One state transition diagram is prepared for each design object.

Design-Service-Package Descriptions - documentation that describes each design service package in terms of its processing, its dependencies, and its contents. One description is prepared for each design service package identified.

3.1.5 Package Design-Object Model

The Design-Object Model is prepared, packaged and reviewed after

- design objects that satisfy the use cases assigned to each package have been identified, described, and appropriately distributed,
- design objects have been adjusted to operate within the implementation environment,
- the ability of the design objects to satisfy the processing requirements of the allocated use cases has been demonstrated,
- all operations required of design objects to support the use cases have been defined, and
- the design objects have been organized into a system architecture of capability layers, subsystems, and packages for the implementation environment. ([6], pp. 31-95)

Design-Object Model (definted in 3.1) - the package of design phase artifacts. It consists of:

- The Design Survey - describes how the system is structured into subsystems, and identifies the design objects of the system. Subsystems and their packages (also a subsystem) are presented in their design hierarchy.
- Traceability from the Domain-Object Model to the Design-Object Model
- Traceability from the Design-Object Model to the Domain-Object Model
- Traceability from the Analysis-Object Model to the Design-Object Model
- Traceability from the Design-Object Model to the Analysis-Object Model
- Use-Case Description: Design
- Interaction Diagrams
- Design-Subsystem Architecture Descriptions
- Design-Service-Package Descriptions
- Attribute-Types: Design - provides a complete list of the attribute types used in the system
- State-Transition Diagrams
- Views - describes specific relationships between packages, objects, actors, use cases and attribute types. Views also are prepared to illustrate specific associations. ([6], pp. 117-138)

3.1.6 Review Design-Object Model Products

The Design-Object Model is reviewed relative to documented completion criteria. ([6], pp. 94-100)

3.1.7 Review Design Activity Completion Conditions

Design activities are reviewed against documented completion criteria. ([6], p. 90)

3.2 Implementation

Each package (design object) is implemented in the target language. ([1], p. 244)

Implementation Model - annotated source code. ([1], p. 149)

4. Testing

Test Model - test specifications and test results for
- unit testing,
- integration testing, and
- system testing. ([1], p. 150, 316)

4.1 Test Planning

Testing requirements are constructed based on the use cases contained in the Requirements Model. ([1], p. 333)

Test Plan - the requirements-based plan for testing the system, which should include unit, integration, and system testing. ([1], p. 333)

4.2 Test Identification

Identify required testing resources and give a detailed estimate of testing requirements. (p. 335)

Resources - any resources allocated to testing. ([1], p. 335)

4.3 Test Specification

Specify each test on the functional level, including a procedure for each step in the test. Prepare documents for recording the results of the tests. ([1], p. 335)

Test Specification - the complete description of all tests to conduct and the information to collect from each, along with criteria for judging the test results. ([1], p. 335)

4.4 Test Execution

Execute the tests. ([1], p. 336)

Test Results - description of failures observed during testing. ([1], p. 336)

4.5 Error Analysis

Determine the fault which caused each failure, and then determine the error which led to each fault. Correct the error and remove the fault. ([1], p. 337)

Analysis of Test Results - the underlying errors which led to failures. ([1], p. 337)

4.6 Test Completion

Restore any equipment and resources allocated to testing. Conduct a review of the test results and errors found. ([1], p. 338)

Testing Lessons Learned - lessons learned from the preparation for and the testing of a software system. ([1], p. 338)

References

[1] Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach, revised 4th printing. Addison-Wesley, 1993.

[2] Objectory AB. Objectory: Project Management, Version 3.6, 1995.

[3] Objectory AB. Objectory: Configuring your Process, Version 3.6, 1995.

[4] Objectory AB. Objectory: Requirements Analysis, Version 3.6, 1995.

[5] Objectory AB. Objectory: Robustness Analysis, Version 3.6, 1995.

[6] Objectory AB. Objectory: Design, Implementation and Testing, Version 3.6, 1995

[7] Objectory AB. Objectory: Introduction, Version 3.6, 1995.

CLEANROOM EXTENSIONS TO THE OBJECTORY PROCESS

The Objectory process is given in the left column as the "base" process, and the Cleanroom process in the right column is mapped to the Objectory process. Comments that straddle the columns are recommended Cleanroom extensions to Objectory. Cleanroom extensions are given immediately after the Objectory activity to which they apply.

OBJECTORY PROCESS	CLEANROOM PROCESS
1. Project Planning	**1. Project Management**
1.1 Objectory Process Configuration *Development Case Description*	*Process References*
1.2 Objectory/Project Management Integration *Tailored Project Management Process*	
1.3 Objectory/Configuration Management Integration *Tailored Configuration Management Process*	
1.4 Project Scoping	
1.5 Project Organization *Project Technical Staffing Requirements*	**1.5 Tasking** *Tasking Records*
1.6 Project Planning *Project Plan*	**1.3 Schedule Development and Maintenance** *Project Schedule*
	1.4 Training
1.7 Development Case Installation *Trained Personnel*	**1.1 Customer Interaction** *Customer Requirements*
	1.2 Process Control *Process Control Standards*
Cleanroom extension: Include Cleanroom Process Control ideas in project management process. Rationale: Objectory has comprehensive instructions for tailoring the process for a software project. Provisions for process control would be a worthwhile addition.	

OBJECTORY PROCESS	CLEANROOM PROCESS
2. Analysis **2.1 Requirements Analysis** *Requirements Model:* *Use-Case Model* *Domain-Object Model* **2.1.1 User (Actor) Identification** *Primary Actor Definitions* *Secondary Actor Definitions* **2.1.2 Specification of Use Cases** *Use-Case Model* **2.1.3 Design Interfaces** *Interface Description* **2.1.4 Identify Problem Domain Objects and Refine** *Domain-Object Model* **2.1.5 Validate Requirements Analysis Products**	**2.1.1 Top-Level Box Structure Specification Development** *Top-Level Box Structure Specification (user interface and black box)*
Cleanroom extension: Prepare a black box specification for the system. Rationale: The top-level black box specification of external behavior identifies all user stimuli and corresponding system responses in all circumstances of use. The black box specification could be prepared from the use-case model, which also identifies external system stimuli and responses. Two advantages associated with a black box system specification are (1) early conceptual integrity (a basis for understanding system parts in the context of the whole), and (2) mathematical rigor (a functional mapping of stimuli histories to system responses).	

OBJECTORY PROCESS	CLEANROOM PROCESS
	2.1.5 Top-Level Usage Specification Development *Usage Specification* **2.1.6 Top-Level Usage Specification Review** **2.4.1 Usage Model Development** *Usage Model* **2.4.2 Usage Model Review and Analysis** *Usage Model Analysis*
Cleanroom extension: Prepare a Cleanroom Markov usage model that identifies the usage states, possible transitions among states, and probabilities of state transitions. Use the model to optimize both development and test planning. Rationale: A system-level Markov usage model is an extension of the Objectory Use Case Model that offers the analytical power of a well-understood formalism. The analytical results can be used to prune the specification, identify development priorities, plan testing, generate test cases, and analyze test results. Cleanroom extension: Use the Markov usage models in certification testing of reusable components. Rationale: If the quality of individual reusable components is to be measured, the Markov usage models derived from the Objectory Use Case Models can be used for test case generation and reliability assessment of components.	
2.2 Analysis Model Development *Analysis Model* **2.2.1 Identify Interface Objects** **2.2.2 Identify Entity Objects** **2.2.3 Identify Control Objects**	**2.1.1 Top-Level Box Structure Specification Development** *Top-Level Box Structure Specification (state box)*

OBJECTORY PROCESS	CLEANROOM PROCESS

Cleanroom extension: Prepare a top-level state box specification for the system.

Rationale: The first activities in Objectory Analysis Model Development are identification of objects that will represent state information. The derivation of the top-level state box from the top-level black box serves exactly this purpose, while preserving the conceptual integrity of the black box specification. For each stimulus-response transaction in the top-level Cleanroom black box, identify information in the stimulus history that is needed to determine the correct response. This information must be maintained in state data objects. The black box specifications may be rewritten as state box specifications:
Black box: (current stimulus, stimulus history) -> (response)
State box: (stimulus, old state) -> (response, new state)

Cleanroom extension: Examine the state box for transaction closure.

Rationale: The Cleanroom principle of transaction closure requires that an object have all the data necessary to process all transactions. The state data must be sufficient to support all transactions, and the stimuli must be sufficient to support all state data.

2.2.4 Subsystem Definition *Subsystems* **2.2.5 Analysis-Object Model Preparation** *Analysis-Object Model*	**2.1.3 Top-Level Box Structure Design Development** *Top-Level Box Structure Design (clear box)* *Common Service Documentation*

Cleanroom extension: Prepare a top-level clear box design for the system.

Rationale: The top-level clear box includes the information in the Objectory Analysis Model describing objects and their associations with other objects. A clear box is derived from its state box:
State box: (stimulus, old state) -> (response, new state)
Clear box: (stimulus, old state) -> (response, new state) via procedure.

The clear box embodies the conceptual integrity of the specification in a top-level architecture. Components of the top-level architecture (i.e., Objectory subsystems) may then be developed in increasing detail.

Cleanroom extension: Ensure referential transparency by preparing a black box specification for each subsystem.

Rationale: The Cleanroom principle of referential transparency requires clearly defined interfaces between an object (in this case, a subsystem) and its user(s). Once an object's function (i.e., the mapping of its stimulus set to its response set) is well-defined, its development may take place independently, without further reference to other parts of the system.

OBJECTORY PROCESS	CLEANROOM PROCESS
2.2.6 Incremental Development Stage Planning *Incremental Development Plan*	**2.2 Increment Planning** **2.2.1 Team Capability and Performance Evaluation** **2.2.2 Increment Plan Development and Maintenance** *Incremental Development Plan*
2.2.7 Validate Analysis Model Products	**2.1.4 Top-Level Box Structure Design Review** **2.3.3 Team Verification** *Usage Hierarchy* *Verification Meeting Records* *Verification Fault Records* *Verified Box Structure Files*

Cleanroom extension: Use the work products in Objectory 2.2.4 and 2.2.7 to prepare a state and clear box for each subsystem.

Rationale: Given earlier.

Cleanroom extension: Ensure referential transparency in each subsystem's clear boxes, by developing black box specifications for next-level subsystems (which may be atomic objects).

Rationale: Given earlier.

Cleanroom extension: Add the rigor of verification to product reviews, including the capture of performance and fault measurements.

Rationale: In Cleanroom team verification, records are kept on faults encountered in each design step. Analysis of performance and fault data provide indicators about process problems.

OBJECTORY PROCESS	CLEANROOM PROCESS
3.0 Construction	**2.3 Box Structure System Decomposition**
3.1 Design	**2.3.1 Individual Specification and Design** *Box Structure Files*
3.1.1 Identify Implementation Environment *Target Environment Description*	
3.1.2 Define Interaction *Interaction Diagram*	
3.1.3 Define Stimuli	
3.1.4 Design Subsystem/Package Architecture *Subsystem/Subsystem Package Architecture*	
3.1.5 Package Design-Object Model	
3.1.6 Review Design-Object Model Products	**2.3.2 Team Specification and Design Review**
	2.3.3 Team Verification *Usage Hierarchy* *Verification Meeting Records* *Verification Fault Records* *Verified Box Structure Files*
3.2 Implementation ***Implementation Model:*** *Pipeline of software increment products* *Documented source code*	**2.3.1 Individual Specification and Design** *Box Structure Files*
Cleanroom extension: Continue the stepwise process of transaction specification (the black box), state data definition (the state box), procedural design (the clear box), and team verification for each level of decomposition. Rationale: A seamless decomposition of specification and design maintains conceptual integrity and satisfies the principles of referential transparency and transaction closure.	

Objectory Process	**Cleanroom Process**
4.0 Testing *Test Model*	**2.4 Statistical Test Planning**
4.1 Test Planning *Test Plan*	**2.4.3 Test Planning** *Test Script*
4.2 Test Identification *Resources*	
4.3 Test Specification *Test Specification*	
4.4 Test Execution *Test Results*	**2.5 Increment Certification** **2.5.1 Compilation** *Compilation Session Records* **2.5.2 Testing** *Annotated Test Script* *Testing Failure Records* **2.5.4 Engineering Change Design** *Compilation Fault Records* *Engineering Change Records* *Updated Box Structure Files* **2.5.5 Engineering Change Verification** *Verified Box Structure Files*
4.5 Error Analysis *Analysis of Test Results*	**2.5.3 Reliability Estimation** *Testing Failure Summary* *Reliability Reports*
4.6 Test Completion *Testing Lessons Learned*	

Cleanroom extension: Certify the system using Cleanroom statistical certification.

Rationale: As a usage-based approach, Cleanroom certification testing is highly complementary with the use-case-driven approach to development in Objectory development. Certification testing provides a statistically valid estimate of reliability in operational use. Statistical testing can be used to certify parts of the system as well as the system in its entirety.

OBJECTORY PROCESS	CLEANROOM PROCESS
1.8 Development Process Improvement *Development Case Improvement Recommendations*	**2.6 Increment Conclusion** **2.6.1 Increment Report Preparation** *Increment Report* **2.6.2 Process Review and Improvement** *Updated Process Control Standards* *Updated Process References*
Cleanroom extension: Review performance in each increment for lessons learned. Rationale: A review of experience at the end of each increment (rather than just at the end of the project) may offer opportunities to improve the development process during a project.	

SHLAER-MELLOR PROCESS - OVERVIEW

In Shlaer-Mellor, domain partitioning drives analysis activity, and domain-specific OOA models are translated to code using generic architectural components ("archetypes"). The Shlaer-Mellor process is the most nontraditional of the processes in this Guide. It is as different from the other OO processes as it is from Cleanroom.

Its strengths relative to other methods are domain partitioning, which is intended to facilitate domain-level reuse; and the archetype concept, which is intended to facilitate translation of analysis models to code.

Its chief differences are the absence of requirements or specifications against which analysis models (the primary Shlaer-Mellor work products) may be validated, and the absence of an explicitly incremental approach to system development and testing.

Of all processes in this Guide, Shlaer-Mellor least emphasizes the management aspects of a software process. The Shlaer-Mellor literature that is currently in the public domain primarily describes the object-oriented analysis method. A book on the Shlaer-Mellor approach to design (called "recursive design") is due out in 1996. The treatment of the Shlaer-Mellor approach as a process in this Guide may have been premature.

A Shlaer-Mellor analysis followed by Cleanroom development may be the best combination of the two processes. A true integration is not likely to be effective due to the strict bottom-up approach in Shlaer-Mellor and the strict top-down approach in Cleanroom.

SHLAER-MELLOR - GLOSSARY

application domain - the subject matter of the system from the perspective of the end user(s) of the system. This is the material that one normally thinks about in the context of requirements analysis: "what does the user need this system to do." ([2], p. 135)

archetype - a built structure that is a combination of code and data, that needs to be completed by adding elements from its client domains such as variable declarations, operation declarations and operation code. ([3], p. 10)

architectural domain - describes generic mechanisms and structures for managing data and control for the system as a whole. The objects in the architectural domain include abstractions of data structures and units of code. Its primary purpose is to impose uniformity on the construction of software. ([2], p. 136)

attribute - an attribute is the abstraction of a single characteristic possessed by all the entities that were, themselves, abstracted as an object. ([1], p. 26)

domain - a world inhabited by its own conceptual entities, or objects, that represents a compositional unit for a large software system. ([2], p. 1)

event - an incident that causes an object to move from one state to another. ([2], p. 5)

implementation domain - describes the conceptual entities in which the entire system will be implemented, e.g. programming languages, networks, operating systems, class libraries, etc. ([2], p. 138)

object - an abstraction of a set of real-world things such that (1) all of the real-world things in the set---the instances---have the same characteristics, and (2) all instances are subject to and conform to the same rules. ([1], p. 14)

relationship - a relationship is the abstraction of a set of associations that hold systematically between different kinds of things in the real world. ([1], p. 47)

service domain - describes generic mechanisms and utility functions as required to support the application domain, and their purpose. ([2], p. 135)

state - a stage in the life cycle for instances of an object. A state model formalizes behavior over time. ([1], p. 93)

subtype object - a specific object with its own attributes and "is a" supertype object with its attributes. ([1], p. 66)

supertype object - a generalizing object which contains the attributes that are common to a group of separate subtype objects. ([1], p. 66)

thread of control - the sequence of actions and events that occurs in response to the arrival of a particular unsolicited event when the system is in a particular state. ([2], p. 94)

References

[1] S. Shlaer and S.J. Mellor. Object-Oriented Systems Analysis: Modeling the World in Data. Prentice-Hall, 1988.

[2] S. Shlaer and S.J. Mellor. Object-Oriented Lifecycles: Modeling the World in States. Prentice-Hall, 1992.

[3] S. Shlaer and S.J. Mellor. "The Shlaer-Mellor Method." Project Technology Technical Report, 1993.

SHLAER-MELLOR - PHASES

A project charter and a requirement specification are prerequisites for performing the Shlaer-Mellor process. From that point, the process spans the development life cycle. Analysis and design activities may be performed independently of one another. The independent development of analysis models and the architectural structures to support their translation into code permit a high degree of concurrency in the design and development of a system, as shown in Figure 3 [5].

The Shlaer-Mellor process assumes the adoption of a suitable project management process to guide a systems development effort and track its progress [4].

1. Analysis

Analysis is concerned with defining the domains describing the problem and solution space for a system and performing a systematic analysis of each domain. The Shlaer-Mellor process assumes a mission statement will be prepared for each domain. A domain mission statement describes the purpose of the domain, the key assumptions for the domain, and the services the domain will provide to and receive from other domains [4].

The product of Analysis is a set of object-oriented analysis models that completely describe the problem space, the solution space, and the functionality for the system. An analysis model is prepared for each domain and each subsystem within a domain. Key to the Analysis phase is the correctness verification of the models produced for each application domain by subjecting the model to:

- static confirmation supported by formal modeling rules, and
- dynamic simulation through the hand execution or simulation of the analysis models [1], [2].

2. Design

Design is concerned with the development of "application-free" generic components in the form of mechanisms, code archetypes and translation rules. Mechanisms are designed and implemented as components that are coded once for an entire system, and serve as a system service, e.g., a timer and an event queue. Archtypes are developed as components that will be generated from information about objects and their operations in the OOA. Translation rules are prepared to map elements of the OOA models, based on the schema of the OOA, into archetypes. For example, an archetype could be prepared to translate object descriptions from an information model to a C++ class, and a translation rule could be prepared that specifies that all objects in the OOA are to be mapped to their corresponding elements in the software architecture, e.g. a C++ class. ([2]; [3], pp. 161-198; [5]).

3. Implementation

Implementation is concerned with producing code for the proposed system. Shlaer-Mellor analysis models are combined with specially designed archetypes (generic code templates) through translation rules. These rules are employed to map elements of the Shlaer-Mellor object-oriented analysis (OOA) models into their architectural equivalents, represented by archetypes. Each object instance in the OOA and its constituent parts (functions and data) and associations will be translated into a corresponding archetype in the architectural domain. The resulting generated source code may be integrated with the mechanisms developed for the system. [5]

References

[1] Lang, N. "Shlaer-Mellor Object-Oriented Analysis Rules," Software Engineering Notes, Volume 18, Number 1, January 1993.

[2] Shlaer, S. and S. Mellor, "The Shlaer-Mellor Method," Project Technology, Inc., Technical Report S075, 1993.

[3] Shlaer, S. and S. Mellor. Object Lifecycles: Modeling the World in States. Yourdon Press, Englewood Cliffs, New Jersey, 1992.

[4] Mellor, S. Personal Communication, 18 February 1996.

[5] Project Technology, Recursive Design: Implementation through Translation - Course Training Materials, Version 1.6. 1994.

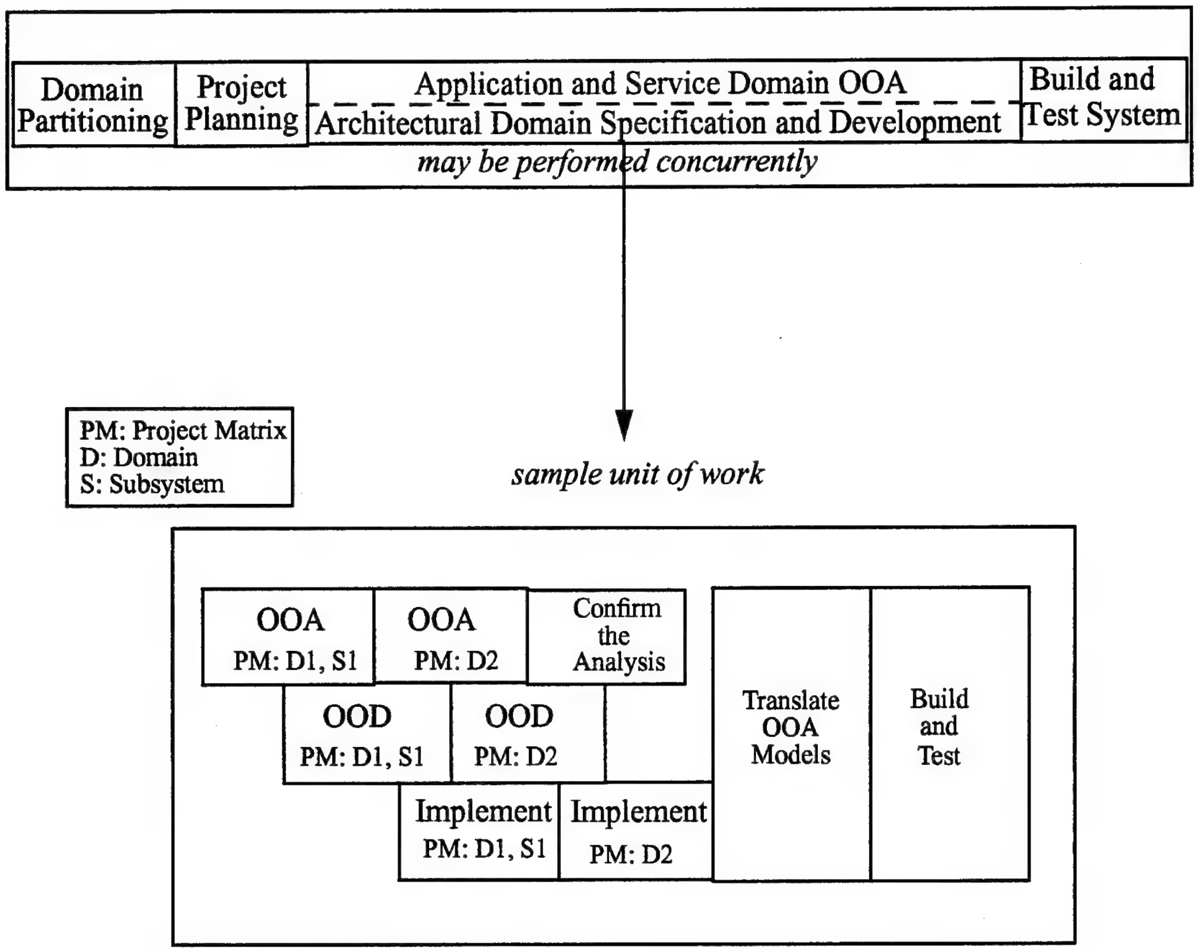


Figure 3: The Shlaer-Mellor Process

SHLAER-MELLOR - ACTIVITIES AND WORK PRODUCTS

The order in which the activities in this section are presented does not imply the order in which they are performed. As described in the discussion of Shlaer-Mellor process phases, the analysis and design phases may be performed independently, allowing a high degree of flexibility in defining units of work.

1. Analysis

1.1 Partition the System into Domains ([1], p. 2)

1.1.1 Identify and Partition Domains of the Proposed System

- Application domains, which describe the system from an end user and system mission perspective.
- Service domains, which will support the application domains.
- Architectural domains, which identify the organization of data, the control mechanisms, and the algorithms for the proposed system.
- Implementation domains, which will support the development and delivery of the proposed system.

Domain Chart - a graphical representation of the domains in which a proposed system will be comprised, and the connections between them. ([4], p. 2)

1.1.2 Prepare Domain Descriptions

Domain Description - A textual description of each of the domains identified on the domain chart. ([1], p. 2; [3], p. 135)

1.1.3 Prepare Domain Bridge Descriptions and Assumptions ([1], pp. 2-3; [3], pp. 134-136)

Cross Domain Bridge Description - A textual description of each bridge between domains identified on the domain chart. ([1], p. 2)

Bridge Assumptions - documented assumptions regarding how the domains identified on the domain chart will interface, how data will be stored and shared, how data will be formatted for presentation to a human or machine agent, etc. ([1], p. 6)

1.1.4 Prepare Project Matrix

Project Matrix - identification of all the domains for a project, and the subsystems of each domain. The project matrix is used to identify cycles for creation of OOA artifacts.

1.2 Analyze the Application Domain ([1], p. 3)

1.2.1 Build an Object Information Model

Object Information Model - a definition of the objects (conceptual entities) of the domain and the relationships between objects.

1.2.2 Build the Object State Model

A state model is built for each object in the object information model that exhibits dynamic behavior or manages contention between other objects.

Object State Model - a Moore machine (deterministic finite state automata) that describes an object's life cycle.

1.2.3 Build the Process Model ([4], pp. 111-132)

The process model describes the processing identified in each state model. One "*action data flow diagram*" is prepared for each state identified in the associated state model's life cycle. Action data flow diagrams show task sequence and conditionality. A *process description* is prepared in an appropriate language for each transformation found on an action data flow diagram.

Process Model - the action data flow diagrams and process descriptions for an object.

1.2.4 Derive Models from the Three Fundamental Models

1.2.4.1 Derive Peer Subsystem Level Models

The following subsystem models are derived to provide an overview of the relationships between subsystems of a domain.

Subsystem Relationship Model - a model of the relationships between objects in different subsystems.

Subsystem Communication Model - a model of asynchronous event communication between objects in different subsystems.

Subsystem Access Model - a model of synchronous data accesses between objects in different subsystems. (Derived from action data flow diagrams.)

1.2.4.2 Derive Object Models that Describe Aspects of the Subsystems

The following object models are derived to provide views and lists that describe characteristics of a subsystem.

Object Communication Model - a model of asynchronous event communication between objects in a subsystem.

Event List - a list of events that are sent within a state model or between state models.

Object Access Model - a model of the synchronous data access between objects within a subsystem.

State Process Table - a list of all processes identified in an action data flow diagram. Analysis of this view sometimes reveals common processes.

Thread of Control Chart - shows sequence of actions executed in response to an external event.

1.3 Confirm the Analysis ([2])

1.3.1 Static Confirmation

Perform static checking of all object-oriented analysis products using the Lang [2] analysis rules.

1.3.2 Dynamic Simulation

Execute object-oriented analysis models to validate they correctly accept a given input and produce the expected result(s).

1.4 External Specification

1.4.1 Define System Boundary

System Boundary Statement - a narrative requirements statement and an external events list. ([1], p. 101)

1.5 Extract Requirements from the Service Domains ([1], pp. 6-7)

Based on the description of the service domain and the assumptions documented in the Domain Bridge Description and Assumptions, define the list of requirements to allocate to the service domains.

Service Domain Requirements - assumptions made by client domains on server domains: qualitative requirements, such as assumptions about the mechanisms that will be provided for communicating with an end user or storing persistent data; and quantitative requirements, such as counts, sizes, timing, and performance.

1.6 Analyze Service Domains ([1], pp. 7-8)

The purpose of this activity is to perform an object-oriented analysis on the service domain(s). Analysis must be performed on each client to understand its requirements before servers can be effectively modeled. The models to be produced are the standard models of a Shlaer-Mellor object-oriented analysis. After the OOA for each service domain is prepared, its results are analyzed and confirmed by following the same activities as in "1.3 Confirm the Analysis."

Information Model, State Model, Process Model, Derived Models (defined above)

2. Design

2.1 Specify the Architectural Domain ([1], pp. 8-9)

The purpose of this activity is to specify the architecture for the problem to be solved. In the Shlaer-Mellor process, the architecture is an application-independent description of the mechanisms and structures for managing data, function, and control for the system as a whole. A Shlaer-Mellor architecture consists of mechanisms (the run-time code of a domain), archetypes and translation rules. Optionally, the architecture may include a translation engine for implementing translation rules and for generating completed archetypes (templates). Archetypes and translation rules are associated with the bridges between application and implementation domains. The resulting architecture provides the blueprint for the integration of mechanisms with the archetypes. The architecture is defined by preparing class diagrams, class structure charts, inheritance diagrams, and dependency diagrams. ([1], p.8)

Class Diagram - the external view of a single class.

Inheritance Diagram - shows the inheritance relationships among classes.

Dependency Diagram - a depiction of the invocation relationships between classes.

Class Structure Chart - the internal structure of the class and its operations.

2.2 Build Architectural Components ([1], pp. 9-10)

The purpose of this activity is to implement the components defined in the architecture. Three types of architectural components are built: mechanisms, archetypes and translation rules. Mechanism components are implemented by designing programs and implementing them in a selected target language. Mechanisms represent services for the system as a whole, e.g., event queues and timers. Archetypes are templates for translating OOA models into source code. They are completed by adding elements from client domain OOAs. Translation rules that define how elements of the OOA models are to be translated into each unique archetype are also defined [5].

Mechanisms - traditional software tasks or library components.

Archetypes - templates for translating OOA models into source code. Archetypes are completed by adding elements (e.g., object instances and the functions and data for each object instance) from OOA models of the application and service domains.

Translation Rules - rules for mapping an element from an OOA model to its counterpart in the software architecture, e.g., an OOA information model object class to a C++ Class.

3. Implementation

3.1 Translate the OOA Models ([1], pp. 10-11)

The purpose of this activity is to translate the object-oriented analysis models into the source code for a target computer language, and to integrate the resulting source code with the mechanism components to develop a software system or the software portion of a system. This translation process involves generating source code components from all elements of the OOA models using archetypes and the translation rules. During this translation process each object instance in the OOA, its constituent parts (functions and data), and its associations with other objects are processed using the appropriate archetype to produce its source code equivalent [5]. Further, where more than one archetype is developed to satisfy an architectural domain processing requirement, a variant of an OOA may be prepared, referred to as a "colored OOA," to specify which archetype will be used in processing an OOA to produce source code components [5].

Populated Task Archetypes - a task archetype that has had its replaceable segments substituted with elements from the OOA models. ([1], p. 11)

Populated Module Archetypes - an archetype implemented to define the structure and contents of the main module for a task in which its associated object or objects would reside. ([1], p. 11)

References

[1] Shlaer, S. and S. Mellor. "The Shlaer-Mellor Method." Project Technology, Inc., Technical Report S075, 1993.

[2] Lang, N. "Shlaer-Mellor Object-Oriented Analysis Rules." *Software Engineering Notes*, Volume 18, Number 1, January 1993.

[3] Carter, C.B., C.H. Raistrick, and K. Carter. "The Shlaer-Mellor Method: A Formalism for Understanding Software Architecture," in Object Development Methods, edited by Andy Carmichael. SIGS Books, New York, 1994.

[4] Shlaer, S. and S. Mellor. Object Lifecycles: Modeling the World in States. Yourdon Press, Englewood Cliffs, New Jersey, 1992.

[5] Project Technology, Recursive Design: Implementation through Translation - Course Training Materials, Version 1.6. 1994.

CLEANROOM EXTENSIONS TO SHLAER-MELLOR PROCESS

The Shlaer-Mellor process is given in the left column as the "base" process, and the Cleanroom process in the right column is mapped to the Shlaer-Mellor process. Comments that straddle the columns are recommended Cleanroom extensions to the Shlaer-Mellor process. Cleanroom extensions are given immediately after the Shlaer-Mellor process part to which they apply.

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
	1. Project Management *Process References* **1.1 Customer Interaction** *Customer Requirements* **1.2 Process Control** *Process Control Standards*
Cleanroom extension: Represent the activities and work products in the Shlaer-Mellor process in a manner that can be used as a "template" for project management. Rationale: While the Shlaer-Mellor Project Matrix (in SM 1.1.4 below) serves as a frame of reference for project information, there is no corresponding structure for process information. A process representation that provides management guidance would specify sequentiality vs. concurrency in process steps and the work products that are used, created, or updated in each process step. This information is provided, but is diffuse in Shlaer-Mellor writing.	

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
1. Analysis **1.1 Partition the System into Domains** **1.1.1 Identify and Partition Domains of the Proposed System** *Domain Chart* **1.1.2 Prepare Domain Description** *Domain Description* **1.1.3 Prepare Domain Bridge Descriptions and Assumptions** *Cross Domain Bridge Description* *Bridge Assumptions* **1.1.4 Prepare Project Matrix** *Project Matrix* **1.2 Analyze the Application Domain** **1.2.1 Build an Object Information Model** *Object Information Model*	**1. Project Management** *Project Records* **1.3 Schedule Development and Maintenance** *Project Schedule* **1.4 Training** **1.5 Tasking** *Tasking Records*
	2.1.1 Top-Level Box Structure Specification Development *Top-Level Box Structure Specification (user interface and black box)*

SHLAER-MELLOR PROCESS	**CLEANROOM PROCESS**
Cleanroom extension: Prepare a black box specification for the system. Rationale: The top-level black box specification of external behavior identifies all user-generated stimuli and the corresponding system responses in all circumstances of use. In the preceding Shlaer-Mellor analysis step, the "originators" of system stimuli were identified. A subsequent system-level specification would map the system stimuli to system responses (at whatever the convenient level of abstraction). The system specification provides early conceptual integrity; it is a basis for understanding system parts in the context of the whole.	
1.2.2 Build the Object State Model *Object State Model*	**2.1.5 Top-Level Usage Specification Development** *Usage Specification* **2.1.6 Top-Level Usage Specification Review** **2.4.1 Usage Model Development** *Usage Model* **2.4.2 Usage Model Review and Analysis** *Usage Model Analysis*
Cleanroom extension: Develop an Object State Model for the system and analyze it as a Markov Chain usage model. Rationale: The Shlaer-Mellor Object State Model is very similar to a Cleanroom Markov Chain usage model. The Shlaer-Mellor literature states that object state models may be combined to create state models for larger objects. If so, Markov analysis of a composite system-level model could provide a wealth of practical information for optimizing development and test resources. In addition, a system-level usage model can be used as a test case generator in statistical testing based on the Markov model. Cleanroom extension: Use the object state models as Markov usage models in certification testing of reusable components. Rationale: If the quality of individual reusable components is to be measured, the Shlaer-Mellor state models for each object can be used for test case generation and reliability assessment in statistical testing based on the Markov model.	
1.2.3 Build the Process Model *Process Model*	

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
	2.1.1 Top-Level Box Structure Specification Development *Top-Level Box Structure Specification (state box)*
Cleanroom extension: Prepare a top-level state box specification for the system. Rationale: The development of the top-level state box supports the identification of objects that may span the conceptual boundary between the Shlaer-Mellor Application Domain and other domains. The state box is derived from the black box. For each stimulus-response transaction in the top-level Cleanroom black box, one identifies information in the stimulus history that is needed to determine the correct response. This information is encapsulated in state data objects. The black box specification: Black box: (current stimulus, stimulus history) -> (response) is rewritten as a state box specification: State box: (stimulus, old state) -> (response, new state). Cleanroom extension: Examine the state box for transaction closure. Rationale: The Cleanroom principle of transaction closure requires that an object have all the data necessary to process all transactions. The state data must be sufficient to support all transactions, and the stimuli must be sufficient to support all state data.	
1.2.4 Derive Models from the Three Fundamental Models **1.2.4.1 Derive Peer Subsystem Level Models** *Subsystem Relationship Model* *Subsystem Communication Model* *Subsystem Access Model*	**2.1.3 Top-Level Box Structure Design Development** *Top-Level Box Structure Design (clear box)* *Common Service Documentation* **2.1.4 Top-Level Box Structure Design Review**

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
Cleanroom extension: Prepare a top-level clear box design for the system. Rationale: The top-level clear box includes the information in the Shlaer-Mellor Subsystem Relationship, Communication, and Access models, and additionally defines the flow of control among subsystems. The clear box: State box: (stimulus, old state) -> (response, new state) is derived from the state box: Clear box: (stimulus, old state) -> (response, new state) via procedure As the top-level system architecture, the clear box preserves the conceptual integrity of the specification from which it was derived. Cleanroom extension: Ensure referential transparency by preparing a black box specification for each subsystem. Rationale: The Cleanroom principle of referential transparency requires clearly defined interfaces between an object (in this case, a subsystem) and its user(s). Once an object's function (i.e., the mapping of its stimulus set to its response set) is defined, its development may take place without further reference to other parts of the system.	
	2.2 Increment Planning **2.2.1 Team Capability and Performance Evaluation** **2.2.2 Increment Plan Development and Maintenance** *Incremental Development Plan*
Cleanroom extension: Prepare an incremental development plan for subsystem development. Rationale: Given a top-level specification, a top-level architecture, and referentially transparent subsystems, the system is now well enough defined to order and schedule development tasks in light of risks and priorities.	
1.2.4.2 Derive Object Models that Describe Aspects of the Subsystems *Object Communication Model* *Event List* *Object Access Model* *State Process Table* *Thread of Control Chart*	**2.3 Box Structure System Decomposition** **2.3.1 Individual Specification and Design Box Structure Files** *Box Structure Files*

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
Cleanroom extension: Use the work products in SM 1.2.4.2 to prepare a state box and clear box for each subsystem. Rationale: Given earlier. Cleanroom extension: Ensure referential transparency in subsystem clear boxes, by developing black box specifications for next-level subsystems (which may be atomic objects). Rationale: Given earlier. Cleanroom extension: Continue the stepwise process of transaction specification (the black box), state data definition (the state box), and procedural design (the clear box) for each level of decomposition.	
1.3 Confirm the Analysis **1.3.1 Static Confirmation** **1.3.2 Dynamic Simulation** **1.4 External Specification** **1.4.1 Define System Boundary** *System Boundary Statement* **1.5 Extract Requirements from the Service Domains** *Service Domain Requirements* **1.6 Analyze Service Domains** *Information Model, State Model, Process Model, Derived Models*	**2.3.2 Team Specification and Design Review** **2.3.3 Team Verification** *Usage Hierarchy* *Verification Meeting Records* *Verification Fault Records* *Verified Box Structure Files*
Cleanroom extension: Use black box stimulus information as an input to Shlaer-Mellor Service Domain Analysis. Rationale: Since service domain objects (e.g., GUI objects) are often sources of external events, they may have been identified and added to the Information Model when the system-level black box was developed.	

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
2. Design **2.1 Specify the Architectural Domain** *Class Diagram* *Inheritance diagram* *Dependency diagram* *Class structure chart*	
Cleanroom extension: Use the box structure hierarchy developed in the analysis phase as the system architecture. Rationale: The need for all but the Inheritance Diagram has been obviated by Cleanroom extension (stepwise unfolding of box structure specification and design) during Shlaer-Mellor Analysis.	
2.2 Build Architectural Components *Mechanisms* *Structures* **3. Implementation** **3.1 Translate the OOA Models** *Populated task archetypes* *Populated module archetypes*	
Cleanroom extension: Implement all clear boxes in the target language.	

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
	2.4 Statistical Test Planning **2.4.3 Test Planning** *Test Script* **2.5 Increment Certification** **2.5.1 Compilation** *Compilation Session Records* **2.5.2 Testing** *Annotated Test Script* *Testing Failure Records* **2.5.4 Engineering Change Design** *Compilation Fault Records* *Engineering Change Records* *Updated Box Structure Files* **2.5.5 Engineering Change Verification** *Verified Box Structure Files* **2.5.3 Reliability Estimation** *Testing Failure Summary* *Reliability Reports*
Cleanroom extension: Certify the system using Cleanroom statistical certification. Rationale: The Shlaer-Mellor method does not prescribe an approach to testing. Cleanroom certification testing is a customer usage-based approach that provides a statistically valid measure of reliability.	
	2.6 Increment Conclusion **2.6.1 Increment Report Preparation** *Increment Report* **2.6.2 Process Review and Improvement** *Updated Process Control Standards* *Updated Process References*

SHLAER-MELLOR PROCESS	CLEANROOM PROCESS
Cleanroom extension: Review performance in each increment for lessons learned. Rationale: A review of experience at the end of each increment of work should be included in the process structure mentioned in the first Cleanroom extension. The process review is the forum for group decisions about process improvement.	

V. Generic Object-Oriented Cleanroom Process

Overview

The Generic Software Process defined in section II is used in section V as a framework for representing the overall relationship among the OO and Cleanroom phases, activities, and work products described in this Guide. The Generic Process is "instantiated" with OO and Cleanroom work products that best accomplish the work to be done in the Generic Process.

In the table identified as "Generic Process using OO and Cleanroom Techniques," the Generic Process is given in the left column, and the OO and Cleanroom processes are given in the rightmost four columns. Each process has previously been described in terms of phases, activities, and work products, and the table represents a mapping of equivalences (bold text), near equivalences (nonbold text), and lack of equivalences (no text).

The shaded portions of the table in the Generic Process column represent the work to be done in a software project. The shaded work products in the other columns represent the best options for accomplishing the work. Finally, comments that straddle the columns provide perspective on the OO and Cleanroom process parts from which the shaded options were chosen.

The table is intended to provide a process roadmap for integration of object-oriented and Cleanroom processes. The user may obtain a high-level perspective on coverage and depth of the processes included in this Guide, and may identify recommended options that have emerged from this study for performing work across the software life cycle.

Generic Process Using OO and Cleanroom Techniques

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
1. Concept Definition	**1. Conceptualiza-tion**	Prestudy; Feasibility study **2.1 Requirements Analysis** ([3], p. 443)	1.1.1 Identify and Partition Domains	
1.1 Define the Mission	Micro Process cycle to implement concept prototype	Develop & evaluate project needs and ideas; ([3], p. 444)	1.1.2 Prepare Domain Descriptions	**1.1 Customer Interaction**
Mission Statement	***Executable Prototype*** ***Vision of the Project's Requirements***	*High level requirements* *Needs statement; ([3], p. 444)*	-	***Customer Requirements (initial)***
There is nothing object-oriented about concept definition. It is an important step, however, in establishing the context for systems analysis. Booch's "Vision of a Project's Requirements" and Objectory and Cleanroom early requirements statements all address concept definition.				
2. Systems Analysis	**2. Analysis**	**2. Analysis**	**1. Analysis**	
2.1 Analyze Problem Domain	**2.1 Domain Analysis**	**2.1 Requirements Analysis**	**1.1 Partition the System into Domains** **1.2.1 Build Object Information Model**	2.1.5 Top-Level Usage Specifica-tion Development
Domain Description	***Domain Model***	***Requirements Model:*** ***Use-Case Model*** ***Domain-Object Model***	***Domain Chart*** ***Domain Description*** ***Object Informa-tion Model***	*Usage Specification*

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
Booch, Objectory, and Shlaer-Mellor all have work products that describe the environment in which a system will operate and the domain objects of that environment. Objectory employs a particularly popular approach for describing "use cases" from which domain objects are identified.				
2.2 Analyze Requirements	**2.2 Scenario Planning** **Perform Micro Process**	**2.1 Requirements Analysis**	**1.2 Analyze the Application Domain** **1.4.1 Define System Boundary** 1.5 Extract Requirements from the Service Domains 1.6 Analyze Service Domains	**1.1 Customer Interaction**
Requirements Statement	***System Contex Description*** ***Scenarios*** ***Function Point Descriptions*** ***Micro Process Work Products***	***Use-Case Model***	***Boundary Statement*** ***Service Domain Requirements*** ***Application and Service Domain OOA Models***	***Customer Requirements (revised)***
Objectory's Use-Case Model is thorough, formal, and in the customer's language. It may be supported by Booch micro-process activities for exploratory prototyping.				
2.3 Plan Specification and Design Activities	**2.2 Scenario Planning**	**1.4 Project Organization** **1.5 Project Planning**	**1.1.4 Prepare Project Matrix**	**1.3 Schedule Development and Maintenance**

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
Specification and Design Plan	***Risk Assessment***	***Project Plan*** ***Project Technical Staffing Requirements*** *Quality Assurance Plan* *Time and Resource Plan*	***Project Matrix***	***Project Schedule***
Although all the methods address planning, Objectory provides the most comprehensive planning recommendations. Objectory relies on the completion of Use-Case analysis to scope the remaining analysis and design effort. A risk assessment should be incorporated in the planning process, as does Booch.				
2.4 **Review Analysis Phase Work Products**	Scenario walkthroughs ([2], p. 108)	**2.1.5.1** **Review Use-Case Model Products** **2.1.5.2** **Review Requirements Analysis Completion Criteria**	**1.3** **Confirm the Analysis**	-
Analysis Review Criteria	*Examine scenarios for completeness, stability and simplicity ([2], p. 107)* *Team signoff of scenarios for all fundamental system behaviors ([2], p. 107)*	***Team approval and customer acceptance of Use-Case Model Products ([6], pp. 61-64, pp. 11-12)***	***Lang OOA Static Analysis Rules*** [11]	-
Booch, Objectory, and Shlaer-Mellor define criteria for examining their analysis products. Objectory's review criteria are the most comprehensive of the three processes for reviewing the results of a requirements analysis.				
3. **System Specification**	**2.** **Analysis**	**2.1** **Requirements Analysis**	1.4 External Specification	**2.1** **Top-Level Specification and Design**

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
3.1 Specify User Interface	**2.2 Scenario Planning**	**2.1.3 Design Interfaces**	**1.5 Extract Require-ments from the Service Domains (Analysis of "Us-er Interface Ser-vice Domain")**	**2.1.1 Top-Level Box Structure Specification Development (user interface specification)**
User Interface Specification	***Scenarios***	***Interface Description***	**Define interfaces: screens, reports, operator proce-dures, interfaces to other comput-ers** **OOA of User Interface Domain**	***Top-Level Box Structure Specification (user interface specification)***
User Interface Prototype(s)	***Executable prototype***	-	-	-
Early specification of the user interface is extremely important. Objectory addresses user inter-face design in requirements analysis, and Cleanroom does so in top-level black box specifica-tion. Booch also suggests that executable prototypes be developed to demonstrate interface concepts to users.				
3.2 Describe Usage Scenarios	**2.2 Scenario Planning**	**2.1.1 User (Actor) Identification** **2.1.2 Specification of Use Cases**	1.2.2 Build Object State Model 1.2.4.2 Derive models that Describe Aspects of the Subsystems	**2.1.1 Top-Level Box Structure Specification Development** **2.4.1 Usage Model Development**
External Stimulus and Response List	***Scenarios***	***Use Cases***	*Event List* *Object State Model*	***Top-Level Box Structure Specification (stimulus and response list)***
Usage Scenarios	***Scenarios***	***Use Case Model***	*Object State Model*	***Usage Model***

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
Booch, Objectory, and Cleanroom call for the development of models that describe usage scenarios. The Cleanroom usage model is formalized as a Markov chain of usage states and transition probabilities between states. Either Objectory use cases or a Shlaer-Mellor system-level object state model could be used to prepare a Cleanroom Markov usage model. As a well-understood formalism, a Markov chain usage model can be analyzed to optimize development and testing resources and can be used as a test case generator.				
3.3 Specify Software System	2.1 Domain Analysis	**2.1 Requirements Analysis**	**1.2.1 Build Object Information Model** **1.2.2 Build Object State Model**	**2.1.1 Top-Level Box Structure Specification Development (black box)**
Software Specification	*System Context Description* *Micro Process Work Products*	***Use-Case Model***	*ObjectInformation Model* *Object State Model*	***Top-Level Box Structure Specification (black box)***
The Objectory Use Case Model and the Cleanroom Box Structure Specification together are excellent (and complementary) approaches for specifying the external behavior of a system. An Objectory Use Case Model may be formalized in a Cleanroom black box functional (mathematical) specification that maps sequences of external stimuli to external responses.				
3.4 Review Specification Phase Work Products	Scenario Walkthrough	**2.1.5.1 Review Use-Case Model Products** **2.1.5.2 Review Requirements Analysis Completion Criteria**	1.3 Confirm the Analysis	**2.1.2 Top-Level Box Structure Specification Review**
Specification Review Criteria	-	***Review team approval and customer acceptance of Use-Case Model Products ([6], pp. 61-64, pp. 11-12)***	*Lang OOA Static Analysis Rules* [11]	***Specification Review Protocol*** ***Usage Model Review Protocol***
Both Objectory and Cleanroom present criteria for reviewing the correctness, consistency, and completeness of specifications. Booch also identifies the importance of validating aspects of the specification for a system by conducting scenario walkthroughs.				

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
4. System Design	**3. Design**	**2.2 Analysis Model Development**	**1.2 Analyze the Application Domain**	**2.1.3 Top-Level Box Structure Design Development**
4.1 Identify Logical System Objects	**Micro process**	**2.2.1 Identify Interface Objects** **2.2.2 Identify Entity Objects** **2.2.3 Identify Control Objects**	**1.2.1 Build an Object Information Model**	**2.1.2 Top-Level Box Structure Design Development (state box)**
Logical Object Catalog	***Object Diagrams***	***Analysis-Object Model (entity, interface, control objects)***	***Object Information Model*** ***Subsystem Relationship Model***	***Top-Level Box Structure Design (state box)***
All methods have activities for modeling logical software objects and their characteristics. Notable approaches to this activity are Objectory, which requires objects to be identified as to role (i.e., interface object, entity object, or control object), and Cleanroom, in which internal state objects are the explicit encapsulation of an object's external stimulus history.				
4.2 Prepare Implementation-Independent Software Design	**3.1 Architectural Planning**	**2.2 Analysis-Object Model Development**	**1.2.3 Build the Process Models** **1.2.4 Derive Models from the Three Fundamental Models**	**2.1.2 Top-Level Box Structure Design Development**

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
Implementation Independent Software Design	***Architecture Description*** ***Micro Process Work Products***	***Analysis-Object Model:*** ***Use-Case Description - Analysis*** ***Analysis Sub-system Description*** ***Analysis Service Package Description***	***Subsystem Relationship Model*** ***Subsystem Communication Model*** ***Subsystem Access Model*** ***Object Communi-cation Model*** ***Object Access Model*** ***State Process Table*** ***Thread of Control Chart***	***Top-Level Box Structure Design (clear box)***
All methods have techniques for implementation independent logical design. Cleanroom further enforces the mathematical properties of referential transparency and functional verifiability. Any of the methods would be reinforced by verification as a Cleanroom Box Structure Design.				
4.3 Develop Software Architecture	**3.1 Architectural Planning** **3.2 Tactical Design**	**3. Construction: 3.1 Design**	**2.1 Specify Architecture Domain** **2.2 Build Architectural Components** Build Model Com-piler	**2.1.3 Top-Level Box Structure Design**
Candidate Software Architecture(s)	*Executable architecture prototypes*	*Architectural trades from Pre-study and Feasibil-ity Study*	-	-
Only Booch and Objectory identify the need for architectural trades.				

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
Software Architecture	***Executable and Baselined Architecture*** *Micro Process Work Products*	***Design-Object Model (Final):*** ***Subsystem/Sub-system Package Architecture*** ***Target Environment Description***	***Class Diagrams*** ***Inheritance Dia-grams*** ***Dependency Dia-grams*** ***Class Structure Diagrams*** ***Specifications for: Task Archetypes and Class Archetypes***	***Top-Level Box Structure Design*** ***Common Service Documentation***
All the methods provide for defining the software architecture. All have formalisms for depicting architectural structures and their relationships. Booch class diagrams, Objectory subsystem/package diagrams and Shlaer-Mellor Class Diagrams and Structure Charts appear equally useful for supporting architectural representations. Cleanroom architecture is represented as the top-level clear box at the highest level and the full box structure hierarchy in complete form. A Cleanroom architectural description has the merit of functional (mathematical) verifiability of designs to specifications throughout the hierarchy. Without further study and the comparison of actual examples, it is difficult to recommend any one of the approaches over others.				
4.4 Specify Subsystems	Perform Micro Process to analyze and specify each software object	**3.1.4 Design Subsystem Packages**	1.2 Analyze Application Domain 1.2.4 Derive Models from the Three Fundamental Models	**2.1.3 Top-Level Box Structure Design Development**
Software Subsystems	***Key Abstraction Roles and Responsibilities Specification*** ***Executable Prototype***	***Design-Object Model:*** ***Use-Case Design Description*** ***Design-Subsystem Description*** ***Design-Service-Package Description***	OOA of Subsystems	***Top-Level Box Structure Design (black box specifications for subsystems)***

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
A Cleanroom black box specification completely specifies an object's external behavior. In an architecture of communicating objects, black box specification of objects (in this case, subsystems) ensures the mathematical principle of referential transparency in system design. (Once an entity is defined as an 8, it may be implemented as (6+2), (7+1), (3+1+4), or any other equivalent of 8 without regard to how the 8 will be used.)				
4.5 Review of Design Phase Work Products	Acceptance Review of System Architecture ([2], p. 127)	**2.2.7 Review Analysis-Object Model Products** **3.1.6 Review Design-Object Model Products**	Formal review mentioned	**2.1.4 Top-Level Box Structure Design Review**
Design Review Criteria	Architectural review - Rules of thumb ([2], p. 128)	**Analysis-Object Model Review Criteria ([5], pp. 129-144)** **Design-Object Model Review Criteria ([6], pp. 94-99)**	***Lang OOA Static Analysis Rules*** **[11]**	***Design Review Protocol***
All of the methods provide some guidance for the review of the implementation-independent design and the system architecture. Of these, Objectory, Shlaer-Mellor, and Cleanroom have well-documented review criteria.				
5. System Implementation	**4. Evolution**	**3.2 Implementation** **4. Testing**	**3. Implementation Phase**	**2.2 Increment Planning** **2.3 Box Structure System Decomposition** **2.4 Statistical Test Planning** **2.5 Increment Certification**
5.1 Plan Increments	**3.3 Release Planning**	**Incremental Development** ([2], pp. 456-459)	1.1.4 Prepare Project Matrix	**2.2 Increment Planning**

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
Increment Development Plan	***Release Plan*** ***Revised Risk Assessment***	***Increment Plan*** ***Test Plan***	-	***Incremental Development Plan***
Booch, Objectory and Cleanroom all address increment planning. Of these, the Objectory and Cleanroom methods are described most fully.				
5.2 Develop Increment	**4. Evolution** **4.1 Application of the Micro Process**	**3.2 Implementation** **4. Testing** (Unit Testing)	**3.1 Translate the OOA Models**	**2.3 Box Structure Decomposition**
Implemented software increment	***Stream of executable releases*** ***Behavioral prototypes*** ([2], p.132) ***System and user documentation***	***Implementation Model*** ***Test Specifications***	***Populated task archetypes*** ***Populated module archetypes***	***Verified Box Structure Files*** ***Usage Hierarchy*** ***Verification Meeting Records*** ***Verification Fault Records***
Objectory and Cleanroom offer the most cohesive approaches to designing and implementing software increments. Shlaer-Mellor uses a "translation" approach, where generic mechanisms and structures in the architecture (referred to as "archetypes") are completed for the application. Overall, Cleanroom appears to minimize the risk of defects in implementation, and Shlaer-Mellor appears to maximize the potential for reuse.				

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
5.3 **Test Increment**		**3.** **Testing**		**2.4** **Statistical Test Planning** **2.4.1** **Usage Model Development** **2.4.2** **Usage Model Review and Analysis** **2.4.3** **Test Planning** **2.5** **Increment Certification** **2.5.1** **Compilation** **2.5.2** **Testing** **2.5.3** **Reliability Estimation** **2.5.4** **Engineering Change Design** **2.5.5** **Engineering Change Verification**
Tested software increment	***Test Criteria (test plan and test script)*** ***Executable Release***	***Test Model:*** ***Use Case Test*** ***System Test:*** ***-Operation Test*** ***-Performance Test*** ***-Breakage Test*** ***-Ergonomic Test*** ***-User Documentation Test*** ***-Acceptance Test*** ***([6], pp. 160-162)***	Tested Programs ([2], p.107)	***Test Script*** ***Compilation Session Records*** ***Annotated Test Script*** ***Testing Failure Records*** ***Reliability Reports*** ***Increment Report***

GENERIC PROCESS	BOOCH	OBJECTORY	SHLAER-MELLOR	CLEANROOM
Cleanroom is the only method to support statistical certification of software. Cleanroom certification models software testing as a statistical experiment yielding a scientifically valid estimate of reliability.				
5.4 Review Increment Work Products	Software Quality Review of Software Products [2]	**3.1.6 Review Design-Object Model Products**	Formal review mentioned	**2.3.3 Team Verification** **2.4.2 Usage Model Review and Analysis** **2.5.3 Reliability Estimation;** **2.6.2 Process Review and Improvement**
Software Increment Review Criteria	***Quality Assurance Results*** Executable Release Acceptance ([1], p. 262)	***Design-Model Review Recommendations ([6], pp. 94-99)*** ***Reviewing Inherited Operations ([6], p. 163)*** ***Reviewing Abstract Classes ([6], pp. 163-164)***	-	***Customer Requirements*** ***Verification Protocol*** ***Usage Model Review Protocol*** ***Process Control Standards*** ***Process References***
Objectory and Cleanroom are the only processes to identify protocols and review criteria for reviewing the results of software increments.				

References

Booch Process & Methods:

[1] Booch, G. Object-Oriented Analysis and Design with Applications, 2nd edition. Benjamin-Cummings, 1994.

[2] Booch, G. Object Solutions: Managing the Object Oriented Project, Addison Wesley, 1996.

Objectory Process and Methods:

[3] Jacobson, I., M. Christerson, P. Jonnsson, and G. Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

[4] Objectory: Project Management

[5] Objectory: Configuring your Process

[6] Objectory: Requirements Analysis

[7] Objectory: Robustness Analysis

[8] Objectory: Design, Implementation and Testing

Shlaer-Mellor Process and Methods:

[9] Shlaer, S. and S.J. Mellor. Object-Oriented Systems Analysis: Modeling the World in Data. Prentice-Hall, 1988.

[10] Shlaer, S. and S.J. Mellor. Object-Oriented Lifecycles: Modeling the World in States. Prentice-Hall, 1992.

[11] Lang, N. "Shlaer-Mellor Object-Oriented Analysis Rules," Software Engineering Notes, Project Technology Technical Report, January 1993.

[12] Shlaer, S. and S. J. Mellor, "The Shlaer-Mellor Method," Project Technology, Inc., Technical Report, 1993.

VI. CONCLUSIONS

Shared Fundamentals

There is really no conflict between object-oriented methods and Cleanroom software engineering on the fundamentals of software parts and wholes. There is broad agreement, for example, that

- objects are defined by (1) their external behavior and (2) their internal data and access programs;

- systems are defined by (1) their external usage scenarios and (2) their internal organization of object accesses; and

- abstraction, decomposition, hierarchy, and other strategies are all important in identifying and relating the parts of a problem.

Furthermore, there is really no more difference between a particular OO method and Cleanroom than there is between particular OO methods---in some cases, perhaps less. There may be more difference between the Objectory and Shlaer-Mellor approaches, for example, than between Objectory and Cleanroom.

Difference in Focus

The OO and Cleanroom communities do, however, focus on different aspects of software quality. The OO community is generally focused on reusability through domain understanding, and the Cleanroom community is generally focused on reliability through process control. Indeed, all do concern themselves with both these aspects of software quality and more, but the aforementioned difference in focus does exist.

This Guide to Integration of Object-Oriented Methods and Cleanroom Software Engineering is not about resolving conflict between OO and Cleanroom, but about identifying the leverage that each can find in an explicit alliance with the other.

OO Leverage for Cleanroom

For those whose base is Cleanroom, the alliance is simple: add the OO Analysis phase activities (in any of the methods in this Guide) to the Cleanroom process prior to Specification. The conceptual models in OO analysis aid in problem understanding and will set the stage for a rigorous Cleanroom specification.

Cleanroom Leverage for OO

For those whose base is an object-oriented method, the alliance with Cleanroom has several points of leverage. The following aspects of Cleanroom are not typically found in OO processes.

- a system-level black box specification
- a black box specification of every object as a mathematical function

- system decomposition under the mathematical principle of referential transparency
- team verification that an object's implementation is a correct realization of the object's mathematical function specification
- use of usage models for statistical test case generation
- statistically valid reliability certification
- process review and improvement

(These points of leverage are covered in greater detail in section IV, in Cleanroom Extensions to each method.)

In general, the most important leverage for OO processes in an alliance with Cleanroom is the application of mathematical function theory in engineering practice. The fundamental idea that an object is a mathematical function---i.e., a mapping of the object's domain (all stimuli in all circumstances of use) to its range (all correct responses)---guides specification (of the function), implementation (as a realization of the specified function), and verification (that the function-as-implemented is equivalent to the function-as-specified). It is the Cleanroom basis in mathematical function theory that affords intellectual control in development and reliable systems in operational use.

The Limits of Integration

Each of the methods addressed in this Guide has its own conceptual foundation.

For Booch, "software growing" occurs through the iterative and opportunistic interplay of macro and micro processes in "round-trip Gestalt design."

In Objectory, a set of "use cases" that define all system behaviors is elaborated to a fully traceable design that is implemented through staged incremental development.

In Shlaer-Mellor, domain partitioning drives analysis activity, and domain-specific OOA models are translated to code using generic architectural components ("archetypes").

In Cleanroom, engineering formalisms underlie incremental development; mathematical formalisms underlie specification, design, and correctness verification; and statistical formalisms underlie certification testing.

Integration (or any other form of combination) of the processes must occur with the same concern for conceptual integrity that must be observed in software product development.

A Personal Perspective

A variety of approaches to integrating OO and Cleanroom are offered in this Guide. In the end, the authors of the Guide have drawn a personal conclusion about the viability of the approaches, and it is offered here:

Cleanroom is more like OO than OO is like Cleanroom. Cleanroom lacks one essential aspect of OO--domain analysis--whereas OO lacks a number of essential Cleanroom characteristics. It would be far easier to add an OO analysis to the front-end of a Cleanroom process than to insert the key Cleanroom characteristics (above) into an OO process. OO analysis followed by a Cleanroom life cycle substantially preserves the benefits of each approach without intrusion or duplication.